Final Master Thesis

LTI ODE-valued neural networks

Adaptation of the BackPropagation algorithm

MEMORY

Author: Albert Prat Baucells

Advisors: Cecilio Angulo Bahón & Manel Velasco Garcia

Date: June, 2018

Escola Tècnica Superior

Enginyeria Industrial de Barcelona

Abstract

In [Velasco et al., 2014], a new approach of the classical artificial neural network architecture is introduced, named 'LTI ODE-valued neural networks', where *LTI ODE* stands for Linear Time Invariant Ordinal Differential Equation. In this novel system, nodes in the artificial neural network are characterized by: inputs in the form of differentiable continuous-time signals; linear time-invariant ordinary differential equations (LTI ODE) as connection weights; and activation functions evaluated in the frequency domain.

It was shown that this new configuration allows solving multiple problems at the same time using a common neural structure. However, the article concludes with the need for developing learning algorithms for the new model of neural network.

Taking as starting point the drawback pointed out in [Velasco et al., 2014], the main objective of this master thesis is to develop a training algorithm for a LTI ODE-valued neural network. As a first and natural approach, modifications of the BackPropagation algorithm is considered as a general framework. Moreover, since the nature of the inputs are differentiable continuous-time signals, it is analyzed how to obtain a model that can be physically implemented in the form of an analogical circuit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The main motivation for the development of this work is to learn about artificial neural networks, in particular, and machine learning, in a general sense. During my master studies I haven't had the possibility to study about these subjects, a field that I find very interesting and driving. When I chose this project my goal was to achieve a solid knowledge about artificial neural networks.

I found this project in particular really interesting because it moves from the classical neural architecture into a new configuration which has the potential of solving multiple problems at the same time using a single neural structure.

This work takes as starting point the article 'LTI ODE-valued neural networks' published in the *Applied Intelligence* journal [Velasco et al., 2014]. In this article a new approach of the classical neural model is introduced. In this new approach, artificial neurons are characterized by: inputs in the form of differentiable continuous-time signals; linear time-invariant ordinary differential equations (LTI ODE) for connection weights; and activation functions evaluated in the frequency domain.

This project focuses on continuing the work of this article.

1.2 Objectives

The main purpose of this master thesis is to develop a training algorithm for a LTI ODE-valued neural network – where *LTI ODE* stands for Linear Time Invariant Ordinal Differential Equation – in order to obtain a model that can be physically implemented in the form of an analogical circuit.

The article in which this project is based leaves some major open questions that need to be developed:

- Establishing the relation between the order of the employed ODE and the maximum

number of solvable problems for a given network configuration.

- Analyzing the possibility to reorder the problems (and the selected frequencies) to minimize the complexity of the artificial neural network.
- Analyzing whether the system's stability can be guaranteed, as well as the convergence to solutions.

During the process of developing a training algorithm for a LTI ODE-valued neural network, these questions will be naturally addressed. To achieve the main goal of the project here are listed the specific objectives:

- Understanding how the classical neural architecture works. In particular, focusing in the training algorithm.
- Analysis of the article in which this project is based in [Velasco et al., 2014].
- Developing a LTI ODE expression than can be used as a connection weight. It will be necessary to ensure stability and the plausibility of implementing the solution in the form of an analogical circuit.
- Once the connections weights of the neural model are addressed, the focus will shift towards the activation function of the neurons evaluated in the frequency domain.
- Once the whole neural model is developed and plausible, a simulation of the neural network will be made using a toy example.
- Analysis of the results and comparison between the final model and the initial approach in [Velasco et al., 2014].

1.3 Methodology and scope of the project

To develop the training algorithm for a LTI ODE-valued neural network a toy example will be used. The training algorithm will be considered successful if it converges in the case of the toy example. It also needs to be taken into account that to consider the LTI ODE-valued model as a success, it should be possible to implement it as an analogical circuit. A further generalization to more complex neural networks will be left open for future studies.

The training algorithm will be developed using the software Matlab. Once the training algorithm will be finished, it will also be programmed using Python.

After the development of the training algorithm using Matlab and Python, the neural network will be simulated as an analogical circuit using the program LTspice with the

learned values obtained during the training process. LTspice is a high performance simulator, schematic capture and waveform viewer designed to make computer simulations of electrical circuits.

1.4 Planning

This work is organized as follows: I will first introduce what an artificial neural network is, and the classical training algorithm, BackPropagation, will be presented.

Afterwards, I will introduce the toy example that will be used during the project to adapt the classical training algorithm to the new configuration.

At this point, this work focuses on the process of finding a LTI ODE working as a weight that meets the requirements of stability and physical plausibility. This is the longest process of the project due to the difficulty of finding stable LTI ODEs that allow for control over the weight values. I will go through the different approaches that have been taken to find suitable stable LTI ODEs.

Once the weight connections are appropriately addressed, I will present our analysis of the results of the training process.

This project ends with the simulation of the toy example using LTspice.

Finally, I will compare the results obtained with the initial proposition of the project. We will see if the open questions left in [Velasco et al., 2014] are answered, as well as new questions will be posed as future research topics.

Chapter 2

Artificial neural networks: state of the art

2.1 Artificial neural networks

Artificial neural networks (ANN), in our case those called feed-forward neural networks or multilayer perceptrons (MLP), are information processing systems inspired by the way biological nervous systems, such as the brain, process information. They are formed by highly interconnected units of calculation called neurons or nodes. The McCulloch and Pitts' neural model is the basic model for explaining how neurons process information in our brain. In this model, each neuron (see Figure 2.1(B)) performs the sum of its inputs and compares the result with a value called bias. The result of this operation is transformed by an activation function, for example a sigmoid, which returns values in a range, mainly between 0 and 1 or between -1 and 1. The result of the activation function is the output of the neuron.

The neurons are so called because of the similarity of these networks with those in the brain (see Figure 2.1(A,C)), but there exists actually a huge difference in complexity between a brain and a ANN ([Sardi et al., 2017]). However, biological realism would impose entirely unnecessary constraints.

Several neurons can be situated in parallel, conforming a layer, and several layers can be connected in cascade, forming a multilayer structure (see Figure 2.1(D)). The outputs of the first layer of neurons are the inputs for the next layer, and the output of the last layer is the final output of the ANN. Each connection between neurons has associated a weight value, so that the input of the activation function of the neuron is the weighted sum of the inputs to the neuron. The first layer of neurons is called input layer. The last layer of neurons is called output layer and neurons in the layers in between are known as hidden units.

In a ANN, a lot of parameters (weights and biases) should need to be tuned. Changing

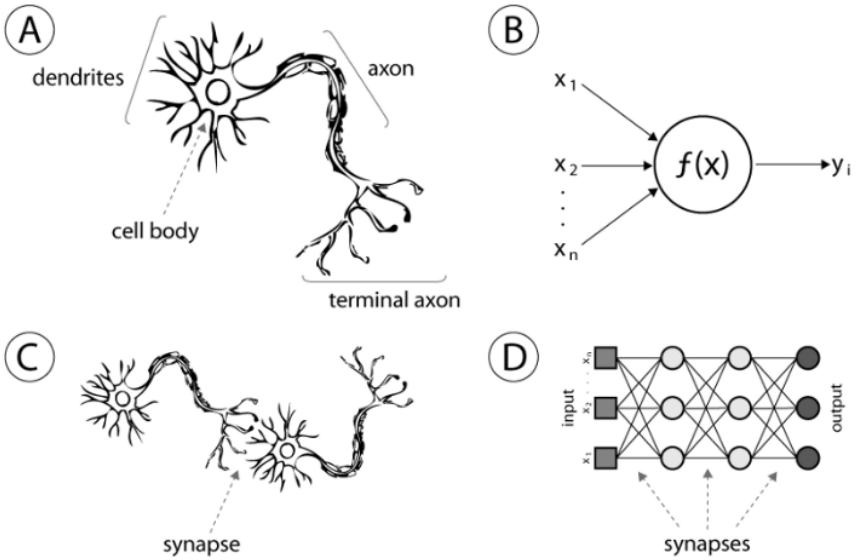


Figure 2.1: Artificial Neural Networks. A comparison between natural and artificial neurons.

the value of these parameters can lead to a huge amount of different outputs. And not only that, we can also play with the number of layers and the number of neurons for each layer. Artificial neural networks provide the possibility of solving an infinite number of tasks with high level of complexity.

So, once we have the structure of the neural network (number of neurons, number of layers, etc.) we need an algorithm that tunes the set of weights and biases that solves the task that we are trying to perform. The BackPropagation algorithm accomplishes this task: if we randomly initialize all the parameters and we compute the output of the ANN we'll find that it is far from the desired output. So we can calculate the difference between the actual output of the ANN and the desired output (called error), and through gradient descent we can minimize this error. Using an iterative procedure of minimization of the error the optimal value for the weights and biases can be found.

2.2 Feed-forward network functions

The feed-forward neural network structure will be implemented according to the Figure 2.2. Neurons are based on linear combinations of fixed nonlinear basis functions $\phi_j(x)$ that take the form,

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(x)\right) \tag{2.1}$$

where $f(\cdot)$ is a nonlinear activation function. The basis functions $\phi_j(\mathbf{x})$ depends on parameters: weights and biases. These parameters are adjusted during training. There are many ways to construct parametric nonlinear basis functions. Neural networks use

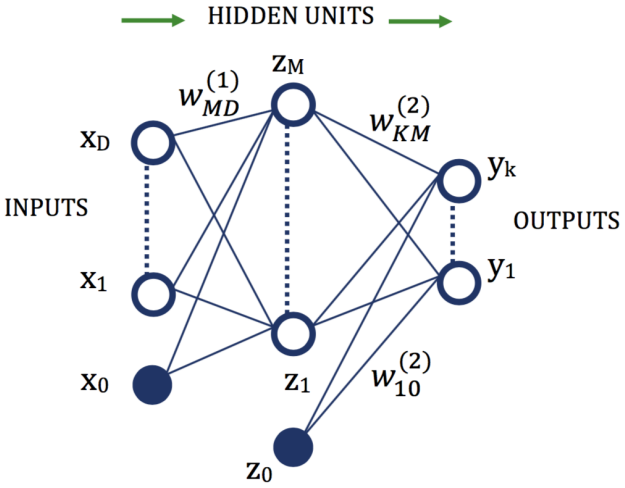


Figure 2.2: Network diagram for the two-layer neural network. The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. Arrows denote the direction of information flow through the network during forward propagation.

basis functions that follow the same form as (2.1), so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

This definition of neural units leads to the basic neural network model. First we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form,

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.2}$$

for $j = 1, \ldots, M$, where the superscript indicates that the corresponding parameters are in the first layer of the network. We will refer to the parameters $w_{ji}^{(1)}$ as weights and the parameters $w_{j0}^{(1)}$ as biases. The quantities a_j are known as activation values. Each of these activation values is then transformed using a differentiable, nonlinear activation function $h(\cdot)$:

$$z_j = h(a_j) \tag{2.3}$$

These quantities, z_j, correspond to the outputs of the basis functions in (2.1). The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the *tanh* function. Following (2.1), these values are again linearly combined to give output unit activations,

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.4}$$

for $k = 1, \ldots, K$, being K the total number of outputs. This transformation corresponds to the second layer of the network, which in the example shown in Figure 2.2 is the output layer of the neural network. Again, $w_{k0}^{(2)}$ parameters are biases. Finally, the output unit activations are transformed using an appropriate activation function σ to give a set of network outputs y_k.

We can combine the stages just explained to give the overall network function:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=1}^{M} w_{kj}^{(2)} h \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \tag{2.5}$$

where the set of all weight and bias parameters have been grouped together into a vector $\mathbf{w}$. So, the neural network model is simply a nonlinear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$ controlled by a vector $\mathbf{w}$ of adjustable parameters.

The bias parameters in (2.2) and (2.4) can be inserted into the set of weight parameters by defining additional input variables x_0 and z_0 whose value is always 1, so that the overall network function becomes,

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=0}^{M} w_{kj}^{(2)} h \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right) \tag{2.6}$$

The network architecture shown in Figure 2.2 can be easily generalized by considering additional layers of neurons. A multi layer network in this form can approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units. The key problem is how to find suitable parameter values given a set of training data. In the next section we will show an effective solution to this problem.

2.3 Network training

2.3.1 Error function and gradient descent minimization

A simple approach to the problem of determining the network parameters is to minimize a sum-of-squares error function. Given a training set comprising a set of input vectors $\{\mathbf{x}_n\}$, for $n = 1, \ldots, N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}$, we minimize the error function

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{t}_n - \mathbf{y}_n(\mathbf{x}_n, \mathbf{w})\|^2 \tag{2.7}$$

We need to find the weight vector $\mathbf{w}$ which minimizes the error function $E(\mathbf{w})$. Let's imagine the error function as a surface sitting over the weight space as shown in Figure 2.3. If we perform a small step in the weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ then the change in the

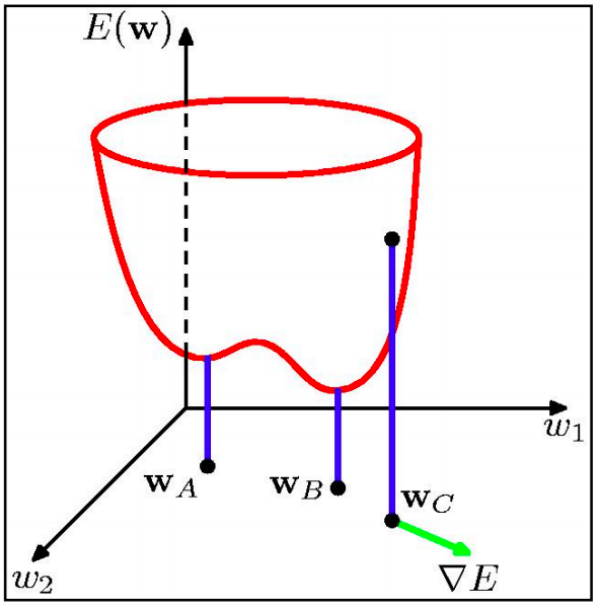


Figure 2.3: Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ΔE.

error function is $\delta E \simeq \delta \mathbf{w}^T \Delta E(\mathbf{w})$, where the vector $\Delta E(\mathbf{w})$ points in the direction of greatest rate of increase of the error function. The error function $E(\mathbf{w})$ is a continuous function of $\mathbf{w}$, so its smallest value will occur at a point in the weight space such that the gradient of the error function vanishes,

$$\Delta E(\mathbf{w}) = 0 \tag{2.8}$$

so we can make a step in the direction of $-\Delta E(\mathbf{w})$ and reduce the error. Our goal is to find a vector $\mathbf{w}$ such that $E(\mathbf{w})$ takes its smallest value. However, the error function typically has a highly nonlinear dependence on the weights and biases parameters, and so there will be many points in the weight space at which the gradient vanishes, or it is very small. A minimum that corresponds to the smallest value of the error function for any weight vector is known as global minimum. Any other minima corresponding to higher values of the error function are known as local minima. It is not necessary for the successful application of neural networks to find the global minimum –and in general it will not be known whether the global minimum has been found. It is enough to find a sufficiently good solution.

It is impossible to find an analytical solution to the equation $\Delta E(\mathbf{w}) = 0$, so we need to use iterative procedures. The simplest approach is the use of the gradient descent optimization algorithm.

The gradient descent algorithm consists in an iterative procedure with adjustments to the weights being made in a sequence of steps. Each step comprises a small step in the

direction of the negative gradient, so that,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \Delta E(\mathbf{w}^{(\tau)}) \tag{2.9}$$

where τ labels the iteration step and the parameter $\eta > 0$ is known as the learning rate. After each such update, the gradient is re-evaluated for the new weight vector and the process is repeated. Note that the error function is defined with respect to the training set, and so each step requires to re-calculate the output of the neural network. Hence, each step of the gradient descent algorithm comprises two stages: in the first stage, known as forward propagation, the output of the neural network for a set of weights $\mathbf{w}$ is calculated. In the second stage, known as BackPropagation, the derivatives of the error function with respect to the weights are evaluated. These derivatives are then used to compute the adjustments to be made to the weight and bias parameters.

In order to find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting vector of weights $\mathbf{w}$.

2.3.2 BackPropagation algorithm

I will now introduce the BackPropagation algorithm, following [Bishop, 2006], for a general two-layer network (see Figure 2.2) using: sigmoidal activation functions and a sum-of-squares error. This example can be easily generalized to a network structure comprised by more than a single hidden layer of neurons.

The error function is a sum of terms, one for each data point in the training set, so that,

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{2.10}$$

In this section, we will consider the problem of evaluating $\Delta E_n(\mathbf{w})$ for one such term in the error function. These derivatives can be used either, directly for sequential optimization, or the results can be accumulated over the entire training set in the case of batch methods. In this project a sequential optimization will be used, so the weights will be updated for each point of the training set iteratively.

For a particular input pattern $\mathbf{x}_n$ in the training set the error function is

$$E_n = \frac{1}{2} \sum_{k=1}^{K} (t_k - y_k)^2 \tag{2.11}$$

where y_k is the output of the output neuron k, and t_k is the corresponding target.

The derivative of the error with respect to the neuron output y_k is

$$\frac{\partial E_n}{\partial y_k} = y_k - t_k \tag{2.12}$$

The neurons have sigmoidal activation functions given by

$$h(a) = \frac{1}{1 + e^{-a}} \tag{2.13}$$

so that the output of each neuron is a value between 0 and 1.

A useful characteristic of this sigmoidal function is that its derivative is very simple,

$$h'(a) = h(a)(1 - h(a)). \tag{2.14}$$

For each pattern in the training set in turn, we first perform a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{2.15}$$

$$z_j = h(a_j) \tag{2.16}$$

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{2.17}$$

$$y_k = h(a_k). \tag{2.18}$$

Now that we have the output of the ANN we can proceed to calculate the derivatives of the error function with respect to the weights using the chain rule,

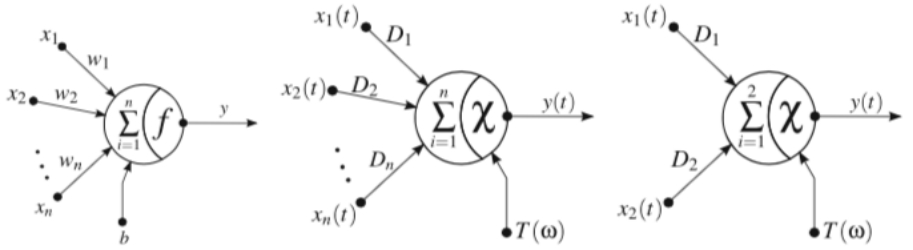
$$\frac{\partial E_n}{\partial w_{kj}^{(2)}} = \frac{\partial E_n}{\partial y_k} \cdot \frac{\partial y_k}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{kj}^{(2)}} = \sum_{k=1}^{K} [(y_k - t_k) \cdot y_k(1 - y_k)] z_j \tag{2.19}$$

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \frac{\partial E_n}{\partial y_k} \cdot \frac{\partial y_k}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}^{(1)}} \tag{2.20}$$

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \sum_{k=1}^{K} [(y_k - t_k) \cdot y_k(1 - y_k) \cdot w_{kj}^{(2)}] \cdot z_j(1 - z_j) \cdot x_i \tag{2.21}$$

Notice that the first two terms of the derivatives of the hidden layer are common with the derivatives of the output layer.

Now that we have all the derivatives calculated we only need to adjust the value of the weights using (2.9).



(a) Classical neuron unit (b) ODE-neuron unit structure (c) ODE-neuron unit example

Figure 2.4: Neuron unit structures. The activation function of the neurons is denoted by $T(\omega)$. The superscripts of the LTI ODE operators have been skipped to simplify the notation.

2.4 The LTI ODE neuron model concept

The LTI ODEVNN (linear time-invariant ordinary differential equation valued neural network) model to be introduced is defined with the same layout as the classical feed-forward network introduced in the previous sections. It will be composed by layers of neuron units connected in cascade, and neurons are still based on the standard neuron shown in Figure 2.4a. The key innovation relies on the fact that the weights of each neuron of a LTI ODEVNN (ODE-neuron) are LTI ODE operators instead of being scalar values in $\mathbb{R}$. As illustrated in Figure 2.4(b,c), LTI ODEs acting as weights, will be referred to as operators and denoted by $D_{ij}^{(l)}$ where the subscript identifies the connection between neurons and the superscript denotes the layer in which the operator is placed.

Inputs will be expressed by a sum of sinus functions, each one characterized by a binary scalar vector of amplitude and frequency. The problems to be solved will be identified each one to a different frequency. The amplitude of the sinusoidal waveform can be identified with the weight value of the classical model of neural networks. From now on, when we talk about weight or bias parameters we will be referring to the amplitude of the signal, whereas if we say operator we will be referring to the LTI ODE functions.

So, by substituting $w_{ji}^{(l)}$ for $D_{ij}^{(l)}$ in all the expressions that define the neural network we can treat the LTI ODEVNN as the classical model. For example, the contribution of the i^{th}-input to the j-ODE-neuron from the first layer can be expressed as

$$a_{ji}(t) = D_{ji}^{(1)} x_i(t), \tag{2.22}$$

so that the result of applying the operator $D_{ji}^{(1)}$ to the input $x_i(t)$ is that each sinus waveform amplitude is modified according to the dynamical response of the LTI ODE operator. Due to the linear nature of the applied operator, the signal $a_{ji}(t)$ is a sum of sinus functions with the same frequency and whose amplitude has been modified by the LTI ODE operator. To put it into another words, if we are trying to solve 3 problems using the same network structure, then, each LTI ODE operator will be codifying 3 weight

values, each one corresponding to one of the sinus signals travelling through the neural network.

The other particularity of the LTI ODEVNNs is the activation function, which needs to apply a sigmoidal function to the amplitude of each sinus signal separately. So that the output of a ODE-neuron will be a sum of sinus signals each one with an amplitude determined by the activation function.

In summary, by specifying m different frequencies appropriately, m problems should be possible to be solved using the same neural network. The issue that needs to be addressed is to find stable and causal LTI ODE operators that allow control over the amplitude of the different sinus signals.

It is important to outline that LTI ODEVNNs can be interpreted as complex-valued neural networks (CVNNs). Complex-valued neural networks deal with complex valued data, complex number weights, and complex valued neuron-activation functions. Their applications have been evolving in various pioneering areas such as electromagnetic-wave and light wave sensing and imaging, independent component analysis in blind separation, or blur restoration in image processing; in engineering areas such as earth and environmental observation with satellite/airborne radar systems, security imaging in public transportation, and medical diagnosis and monitoring. CVNNs are very useful in real-world applications when information is represented by waves such as acoustic, light, or electromagnetic, because of the proper treatment of complex-amplitude information.

In [Velasco et al., 2014] it is proven that for each frequency that characterizes a given input signal, a CVNN is obtained. This means that a LTI ODEVNN that solves m problems can be associated to m CVNNs.

2.5 Toy example

A toy example will be used during the project to develop the training algorithm and to find appropriate LTI ODE operators. The structure of the neural network used, as illustrated in Figure 2.5, is: a two-layer neural network with two inputs, two hidden neuron units and one output neuron unit.

This LTI ODEVNN will be trained for simultaneously solving the XOR, OR and AND Boolean functions. In Table 2.1, some examples of inputs vs expected outputs are shown. Three simultaneous problems are considered. For each problem the same input-output structure based on 3 signals is determined: input 1, input 2 and output of the LTI ODEVNN. Each signal is composed by 3 sinus waveforms of 3 different frequencies f_1, f_2, f_3 codifying each one of the three considered problems (XOR, OR, AND). A value of 1 means the presence of a sinus waveform, and a value of 0 means its absence.

In the next chapters the process of finding an appropriate LTI ODE operator will be explained. As a way of following the process of development of a LTI ODE operator, it

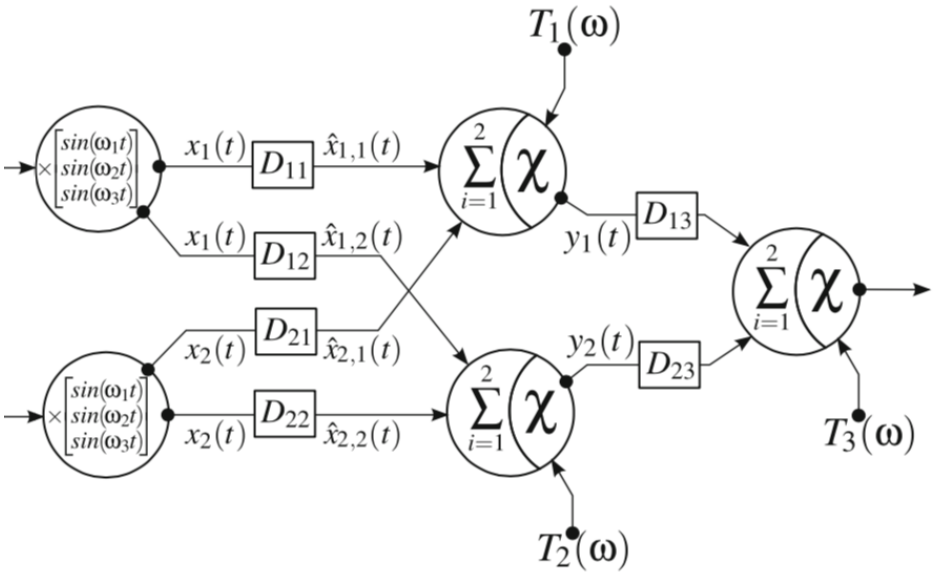


Figure 2.5: Toy example. The activation function of the neurons is denoted by $T(\omega)$. The superscripts of the LTI ODE operators have been skipped to simplify the notation.

is shown in Table 2.2 what we want to achieve at the end of the project: a LTI ODE operator which is stable, physically plausible to implement and able to simultaneously solving 3 problems. During the next chapters, the same table will be expanded in order to keep up with the progress done.

	Input 1	**Input 2**	**Output**
f_1 **(OR)**	1	1	1
f_2 **(AND)**	1	1	1
f_3 **(XOR)**	1	1	0
f_1 **(OR)**	1	0	1
f_2 **(AND)**	1	0	0
f_3 **(XOR)**	1	0	1
f_1 **(OR)**	0	0	0
f_2 **(AND)**	0	0	0
f_3 **(XOR)**	0	0	0

Table 2.1: Examples of inputs vs expected outputs.

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-

Table 2.2: Evolution of the LTI ODE operator: expected results.

Chapter 3

LTI-ODEVNN: LTI ODE operators

3.1 Introduction

As we have seen, the purpose of this project is to simultaneously solve several problems using a single network structure. In this new approach, artificial neurons are characterized by:

- Inputs are coded in the form of differentiable continuous-time signals.
- Linear time-invariant ordinary differential equations (LTI-ODE) encode connection weights.
- Activation functions evaluated in the frequency domain.

In particular, inputs will be expressed by a sum of sinus functions of different frequencies. By applying a LTI-ODE operator to a sinus function, we will be able to change its amplitude and phase. Hence, by carefully choosing adequate weight functions we will determine the final input to a neuron. Then, the neuron will evaluate the activation function for each frequency and the output of the neuron will be the sum of the same sinusoidals but with a new amplitude determined by the activation function. This signal will be the input for the next layer of neurons.

The dynamical response in the frequency domain of a LTI ODE operator (which can be viewed in a bode plot) returns a complex value for each given frequency. A complex value can be written as $g_{\angle\phi}$, where g is the gain and ϕ is the phase. What this means is that if we apply a LTI ODE operator to a sinus function of a given frequency ω_k and amplitude A_k, the result will be a sinus function of the same frequency ω_k but with amplitude $g_k A_k$ and phased ϕ rad with respect the initial sinus function.

So, the first issue that needs to be addressed is to find a suitable, stable and causal LTI ODE operator that allows control over the gain and phase in 3 different frequencies so we can change the amplitude of the sinus signals to the values that we want.

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-
Polynomial	No	No	3	$a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0$

Table 3.1: Evolution of the LTI ODE operator: polynomial approach.

LTI ODEs operators will be expressed as polynomials or quotient of polynomials in the form of Laplace transfer functions using the variable s. To obtain the dynamical response of a LTI ODE operator for a given frequency ω_k, the variable s will be replaced by the isochronous transfer function $j\omega_k$.

3.2 LTI-ODE operator: polynomial approach

An LTI ODE operator, in order to be causal and physically possible to implement in the form of an electrical circuit it needs to be a quotient of polynomials in which the degree of the denominator is higher than the degree of the numerator. A first approach of finding such a LTI ODE operator is to find a single polynomial that for 3 chosen frequencies returns the complex values desired. And only afterwards, to find the quotient of polynomials equivalent.

To have control over the gain and the phase in three different frequencies means that we need 6 parameters. So the polynomial needs to be at least of degree 5,

$$Y(s) = a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0 \tag{3.1}$$

where $Y(s)$ is a complex number.

Substituting s with $j\omega_i$, where ω_i indicates the frequency for each problem, we obtain 6 equations: 3 for the real part of $Y(s)$ and 3 for the imaginary part of $Y(s)$. We have a linear system of 6 equations and 6 unknown coefficients. By solving this system of equations we obtain the set of coefficients that for the 3 given frequencies return the desired complex values.

In the Appendix A, it can be found the Matlab code that finds the coefficients of the polynomial that passes through the 3 desired complex values using a gradient descent algorithm.

Now that we have a single polynomial which returns the desired complex values we want to find the equivalent quotient of polynomials. This task is not trivial, this is why the next approach consists in directly finding a quotient of polynomials (a transfer function) so that the dynamical response in 3 given frequencies can be controlled.

In table 3.1 we can see where we are at the process of finding a suitable LTI ODE operator.

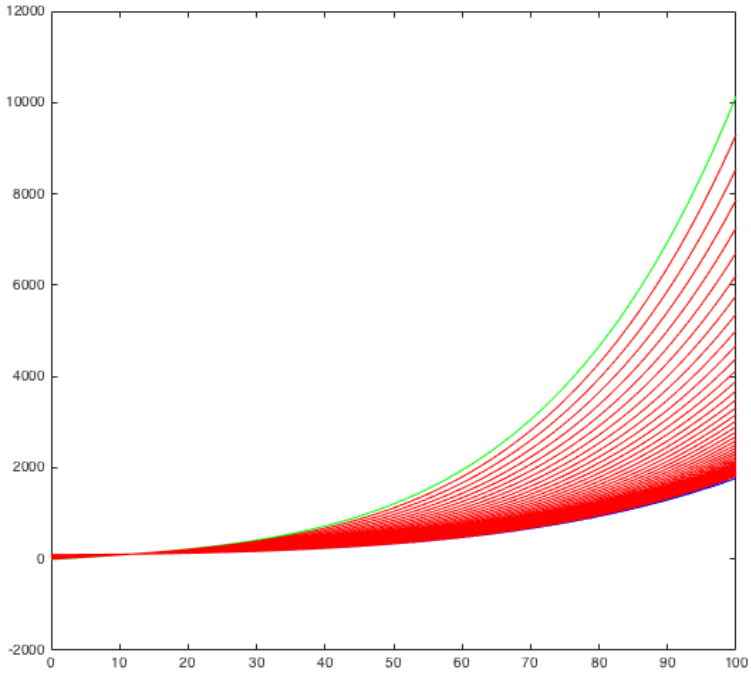


Figure 3.1: Polynomial approach: gradient descent.The green line is the initial polynomial. The red lines are the polynomials found at each iterative step and the blue line is the objective polynomial function (the one that passes through the 3 desired complex values).

3.3 LTI-ODE operator: rational polynomial approach

The objective is to find a quotient of polynomials with the degree of the denominator higher than the degree of the numerator so that the resulting transfer function is causal. A possible transfer function can be:

$$Y(s) = \frac{b_2 s^2 + b_1 s + b_0}{s^3 + a_2 s^2 + a_1 s + a_0} \tag{3.2}$$

which for three given frequencies ω_i, $i = 1, 2, 3$, when we substitute s with $j\omega_i$ it returns 3 desired complex values.

In summary, given 3 points as complex numbers with real part α and complex part β we need to find a quotient of polynomials with real coefficients which passes through these points. In order to do this, let's first substitute $j\omega$ in (3.2) and write $Y(j\omega)$ as a complex number:

$$Y(j\omega) = \alpha + j\beta = \frac{-b_2\omega^2 + jb_1\omega + b_0}{-j\omega^3 - a_2\omega^2 + ja_1\omega + a_0} \tag{3.3}$$

Now let's rewrite this expression:

$$(\alpha + j\beta)(-j\omega^3 - a_2\omega^2 + ja_1\omega + a_0) = (-b_2\omega^2 + jb_1\omega + b_0) \tag{3.4}$$

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-
Polynomial	No	No	3	$a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0$
Rational polynomial	No	Yes	3	$\frac{b_2s^2+b_1s+b_0}{s^3+a_2s^2+a_1s+a_0}$

Table 3.2: Evolution of the LTI ODE operator: rational polynomial approach.

and separate the real terms from the imaginary terms:

$$(-\alpha a_2\omega^2+\alpha a_0+\beta\omega^3-\beta a_1\omega)+j(-\alpha\omega^3+\alpha a_1\omega-\beta a_2\omega^2+\beta a_0) = (-b_2\omega^2+b_0)+j(b_1\omega) \quad (3.5)$$

We can observe that we now have a set of two equations: one with the real part and another one with the imaginary part. Given 3 points in the complex plane $z_i = (\alpha_i + j\beta_i)$ and 3 frequencies $\{\omega_1, \omega_2, \omega_3\}$ we now have 6 equations and 6 unknown coefficients $\{a_0, a_1, a_2, b_0, b_1, b_2\}$, which is a linear system of equations with a unique solution.

So we can find the value of the set of coefficients organized in the form of (3.2) that give a transfer function that passes through 3 given points in the complex plane.

The problem with this approach is that we cannot guarantee the stability of the LTI ODE operator. Moreover, in most of the cases it won't be stable, and we want the weight function to be stable to be able to physically implement it. In the following sections, I will introduce different approaches to find stable LTI ODE operators.

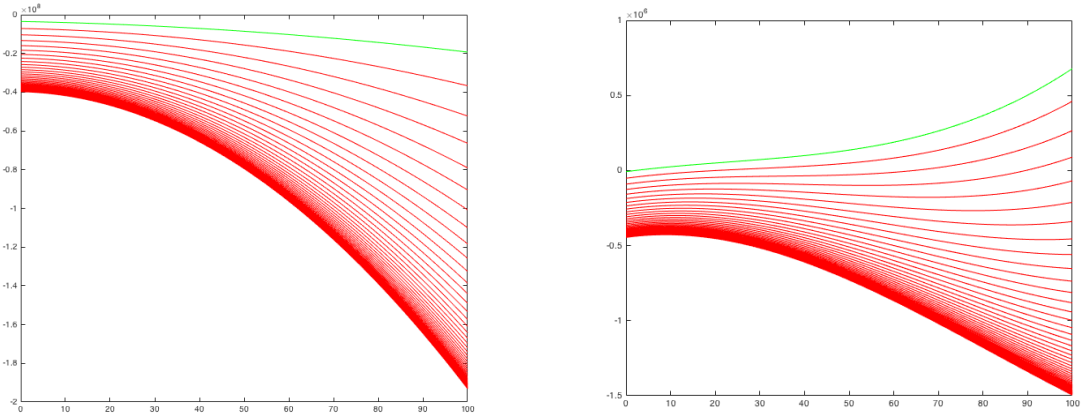
In Appendix A, it can be found the Matlab code for the rational polynomial approach. In the example showed in the Appendix, the denominator of the LTI ODE found is:

$$d = s^3 - 246.3999s^2 + 4.1457e + 03s - 4.4552e + 05 \quad (3.6)$$

which has the following roots:

```
1.0e+02 *
   2.3684 + 0.0000i
   0.0478 + 0.4311i
   0.0478 - 0.4311i
```

They all have positive real parts, so it is unstable. This Matlab code also finds the coefficients using the gradient descent algorithm. In Figure 3.2 it can be seen separately how the numerator and the denominator converge to the solution and in table 3.2 the progress done so far is shown.



(a) Numerator

(b) Denominator

Figure 3.2: Rational polynomial approach.

3.4 LTI-ODE operators: stability

In this section we will design a LTI ODE operator taking into account that it needs to be stable. I will introduce a simpler transfer function and we will analyse whether stability is possible or not.

At this point we will see whether is possible to simultaneously solve 3 problems with a LTI ODEVNN while maintaining the stability of the LTI ODE operator.

To obtain a stable transfer function we need to simplify our requirements. A simple way to do so is to target only the gain of the LTI-ODE operators. This means only focusing on the gain at 3 given frequencies and let the phase take any value.

Hence, first of all let's design a stable transfer function that allows us to choose its coefficients in order to have a desired gain in 3 given frequencies.

For a transfer function to be stable, the denominator needs to satisfy the following conditions:

- All coefficients need to have the same sign.
- All coefficients must have a value different from 0.

These conditions can be studied through the Routh criterion of stability.

It's easy to see that the simplest the denominator is, the simpler will be the stability conditions, because we will have less coefficients. A solution could be to put the coefficients in the numerator to keep the denominator simple. The problem with this configuration is that the physical implementation of a function with a high degree polynomial in the numerator is complicated.

So, if we want to have control over 3 parameters (3 gains) we need at least 3 coefficients

in the denominator. The simplest function that satisfies these conditions is:

$$Y(s) = \frac{1}{as^2 + bs + c} \tag{3.7}$$

Now, let's substitute s with $j\omega$ and find the expression of the gain,

$$|G(j\omega)| = \left|\frac{1}{aj\omega^2 + bj\omega + c}\right| = \left|\frac{1}{(c - a\omega^2) + j(b\omega)}\right| \tag{3.8}$$

$$|G(j\omega)| = \frac{1}{\sqrt{(c - a\omega^2)^2 + (b\omega)^2}} = \frac{1}{\sqrt{a^2\omega^4 + (b^2 - 2ca)\omega^2 + c^2}} \tag{3.9}$$

The final expression is

$$\frac{1}{|G(j\omega)|^2} = a^2\omega^4 + (b^2 - 2ca)\omega^2 + c^2 \tag{3.10}$$

Through Lagrange polynomials, the expression (3.10) can be used to calculate the value of the coefficients (a, b, c) and to analyse the stability of the resulting function,

$$a = \sqrt{\frac{g_1}{(\omega_1^2 - \omega_2^2)(\omega_1^2 - \omega_3^2)} - \frac{g_2}{(\omega_1^2 - \omega_2^2)(\omega_2^2 - \omega_3^2)} + \frac{g_3}{(\omega_1^2 - \omega_3^2)(\omega_2^2 - \omega_3^2)}} \tag{3.11}$$

$$c = \sqrt{\frac{\omega_2^2\omega_3^2 g_1}{(\omega_1^2 - \omega_2^2)(\omega_1^2 - \omega_3^2)} - \frac{\omega_1^2\omega_3^2 g_2}{(\omega_1^2 - \omega_2^2)(\omega_2^2 - \omega_3^2)} + \frac{\omega_1^2\omega_2^2 g_3}{(\omega_1^2 - \omega_3^2)(\omega_2^2 - \omega_3^2)}} \tag{3.12}$$

$$b^2 - 2ca = \frac{g_2\omega_1^2 + g_2\omega_3^2}{(\omega_1^2 - \omega_2^2)(\omega_2^2 - \omega_3^2)} - \frac{g_1\omega_2^2 + g_1\omega_3^2}{(\omega_1^2 - \omega_2^2)(\omega_1^2 - \omega_3^2)} - \frac{g_3\omega_1^2 + g_3\omega_2^2}{(\omega_1^2 - \omega_3^2)(\omega_2^2 - \omega_3^2)} \tag{3.13}$$

where ω_i are the frequencies and g_i the gain at each frequency.

If we look closely at these expressions we can see that we need the expression inside the square root of coefficients a and c to be positive. If we choose carefully the frequencies ω_i and the gains g_i we can make these two coefficients positive. The problem comes with coefficient b. By making the coefficients a and c more positive, b becomes more negative.

In order to be sure if it is possible to make all three coefficients positive I have written a code in matlab which tries to find a solution to the following equation:

$$b^2 = \frac{g_2\omega_1^2 + g_2\omega_3^2}{(\omega_1^2 - \omega_2^2)(\omega_2^2 - \omega_3^2)} - \frac{g_1\omega_2^2 + g_1\omega_3^2}{(\omega_1^2 - \omega_2^2)(\omega_1^2 - \omega_3^2)} - \frac{g_3\omega_1^2 + g_3\omega_2^2}{(\omega_1^2 - \omega_3^2)(\omega_2^2 - \omega_3^2)} + 2ca > 0 \tag{3.14}$$

I use Matlab's `solve` function and I allow the frequencies ω_i and the gains g_i to take any real value. This operation doesn't return any solution, so this approach is a dead end.

So, to find a stable LTI ODE operator we need to simplify even more our requirements

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-
Polynomial	No	No	3	$a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0$
Rational polynomial	No	Yes	3	$\frac{b_2s^2+b_1s+b_0}{s^3+a_2s^2+a_1s+a_0}$
1st stable approach	No	Yes	3	$\frac{1}{as^2+bs+c}$

Table 3.3: Evolution of the LTI ODE operator: first stability approach.

so that the denominator of the transfer function can be simpler and the stability conditions less restrictive. The only way left of simplifying the LTI ODE operators is to attempt to solve 2 problems instead of 3. In the next section we will see if by diminishing the number of problems to be solved by the LTI ODEVNN from 2 to 3 can we find stable LTI ODE operators.

In table 3.3 we can see that until now, we haven't found a stable LTI ODE operator.

In the Appendix, there is another approach that has been tried before reducing the number of problems to be solved. It didn't work, and it is not relevant enough to appear here.

3.5 LTI ODEVNN: 2 problems

Previously, we have seen that if we want to have control over the gains in the 3 given frequencies, we end up with an unstable transfer function. The stability conditions grow quickily more complicated as the number of coefficients increase. So, let's try to build the transfer function with two coefficients instead of with 3. This way, we will have control over the gain in two frequencies and we will be able to solve two problems simultaneously.

So, our transfer function will have the following form,

$$Y(s) = \frac{1}{as + b} \tag{3.15}$$

Now, let's substitute s with $j\omega$ and find the expression of the gain,

$$|G(j\omega)| = \left|\frac{1}{aj\omega + b}\right| = \left|\frac{1}{b + ja\omega}\right| \tag{3.16}$$

$$|G(j\omega)| = \frac{1}{\sqrt{(a\omega)^2 + b^2}} = \frac{1}{\sqrt{a^2\omega^2 + b^2}} \tag{3.17}$$

The final expression is

$$\frac{1}{|G(j\omega)|^2} = a^2\omega^2 + b^2 \tag{3.18}$$

Now, we can find the expressions for the coefficients a and b:

$$a = \sqrt{\frac{y_2 - y_1}{w_2^2 - w_1^2}} \tag{3.19}$$

$$b = \sqrt{\frac{y_1 w_2^2 - y_2 w_1^2}{w_2^2 - w_1^2}} \tag{3.20}$$

where $y_i = \frac{1}{|G(j\omega_i)|^2}$.

To make the transfer function stable in any situation, we need to restrict the possible values for y and ω. A good solution is to make the sinus or cosinus of the gain. This way, we can restrict y to a small range of positive values (for example from 2π to π) and obtain values from -1 to 1. Let's discuss the stability of the transfer function. The condition necessary for stability is that the coefficients a and b are positive.

- Coefficient a: If we consider ω_2 bigger than ω_1, then for a to be positive $y_2 - y_1$ needs to be positive. This means than the gain in frequency ω_2 needs to be smaller than the gain in frequency ω_1.
- Coefficient b: By making the gain in the second frequency smaller than the gain in the first frequency we can be sure that a will be positive. To ensure stability, we need to compute the difference $\delta = y_1\omega_2^2 - y_2 w_1^2$ which indicates whether coefficient b will be positive too or not.

Let's define the gain ranges for each problem. To choose the gain ranges we need to take into account two things:

- The minimum range should not include 0, because gain 0 is $-\infty$ in dB.
- The gain ranges should not overlap each other.

With these two conditions in mind we can choose the following ranges:

- Gain range associated to ω_1: 2π to 3π. The cosinus of this range returns values from 1 to -1.
- Gain range associated to ω_2: $\frac{\pi}{2}$ to $\frac{3\pi}{2}$. The sinus of this range returns values from 1 to -1.

We can now compute the difference δ for any pair of chosen frequencies to check the stability. In order to ensure stability for every possible combination of y_1 and y_2, we can compute the difference δ in the worst case scenerio: when y_1 takes the minimum value and y_2 the maximum value. For the frequencies $\omega_1 = 100Hz$ and $\omega_2 = 1000Hz$, $\delta = 7.205061947899574e + 03$. So, it is proven that for the frequencies and gain ranges chosen all weight functions will be stable.

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-
Polynomial	No	No	3	$a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0$
Rational polynomial	No	Yes	3	$\frac{b_2s^2+b_1s+b_0}{s^3+a_2s^2+a_1s+a_0}$
1st stable approach	No	Yes	3	$\frac{1}{as^2+bs+c}$
2nd stable approach	Yes	No	2	$\frac{1}{as+b}$

Table 3.4: Evolution of the LTI ODE operator: second stability approach.

Now, we can obtain any value in the interval $[-1, 1]$. But this is a strong limitation for a neural network.

Using Matlab, I have programmed two neural networks that solve the AND and XOR Boolean functions separately. In both cases, the final learned weights were values greater than 1. So it would be interesting to find a way to expand the possible range of values outside the interval $[-1, 1]$.

The solution that I have implemented is to associate a value p multiple of 10 to each weight $x \in [-1, 1]$ so the product px returns any real number.

In table 3.4 it is shown that by reducing the number of problems to be solved we have found a stable LTI ODE operator.

3.5.1 AND + XOR Neural Network

In this section I will explain how does the neural network that solves simultaneously the AND and XOR Boolean functions works and I will present one possible set of LTI ODE operators and biases that solve these two problems using the network architecture from the toy example.

In the Annex A there is the code of all the scripts in Matlab used to find the coefficients of the LTI ODE operators and the biases. The different scripts are:

- `main.m`: in this file we first define the frequency associated to each problem. In our case, the frequencies chosen are: 100 Hz for the AND problem and 1000 Hz for the XOR problem. Then the stability is checked computing the difference δ explained in the previous section. Afterwards, I introduce the training data to be used to train the ANN and randomly initialize the values of the biases and coefficients of the weight functions. Then, the iterative learning process begins. This process has two stages: forward propagation and back propagation.

- `weight.m`: this file is a mMtlab function that takes as argument the coefficients of a weight function and the associated p values. The frequency response of the given weight function is evaluated at each frequency. Once the gain at each frequency is obtained this Matlab function performs the sinus of the gain and multiplies the

result by its associated power of 10 (p) in order to calculate the final weight value that will be applied to the input of the neuron.

- `activation.m`: this file is a Matlab function that takes as argument the weighted sum of the inputs to a neuron (z) and returns the output of the neuron. It returns 2 values; one for each frequency. The activation function used is the sigmoidal function $\frac{1}{1+e^{-z}}$.

- `ab.m`: this file is a Matlab function that takes as argument the desired weight values and calculates the coefficients of the LTI-ODE operator that at the chosen frequencies have the desired gain.

- `forwardpropagation.m`: this file performs the weighed sum of the inputs for each neuron and calculates the final output of the neural network.

- `backpropagation.m`: this file calculates the derivatives of the error for each weight and adjusts the weight values and the biases. It returns the new coefficients of the LTI-ODE functions.

Now I will present a possible solution for this neural network. Each time that the program is executed it returns a different possible set of weights and biases because it depends on the initialization, which is random.

The following values are the output of the neural network after being trained for 1000 iterations. The first two columns are the inputs and the last column is the output of the neural network. It can be seen that for both problems the neural network converges and finds a solution.

```
r_and =
         0         0    0.0004
         0    1.0000    0.0499
    1.0000         0    0.0386
    1.0000    1.0000    0.9277


r_xor =
         0         0    0.0575
    1.0000         0    0.9360
         0    1.0000    0.9356
    1.0000    1.0000    0.0812
```

The LTI ODE operators and biases learned are the ones showed next. Each one of the matrix of values listed below corresponds to each neuron. The first row of the matrix are the biases of the neuron: the first column is the bias for the first frequency and the

second column the bias associated to the second problem. The two following rows are the coefficients of the LTI ODE operators. If we consider a transfer function in the form: $\frac{1}{as+b}$ then the values in the first column are the coefficient a and the values in the second column are the coefficient b. The second row is the LTI ODE operator that connects the neuron with the first input and the third row is the LTI ODE operators that connects the neuron with the second input.

To make it more understandable, the following matrix shows how the values are distributed:

$$\begin{bmatrix} bias_{100Hz} & bias_{1000Hz} \\ a_{input1} & b_{input1} \\ a_{input2} & b_{input2} \end{bmatrix}$$

```
d1 =
    3.373786217999898   2.002456303839134
    0.000038540119613   0.117799346387133
    0.000038124015468   0.122783663429689


d2 =
    1.419541739814078   5.734930211949614
    0.000039886648999   0.130225613196706
    0.000040909532626   0.116783084486172


d3 =
    3.325870014907955  -3.239351049607623
    0.000035422708427   0.113838777810037
    0.000065465244283   0.110942757895436
```

Although the values for the coefficient a are very small, these LTI-ODE operators are physically possible to be implemented. In Figure 3.3 it is shown an electrical circuit using an operational amplifier, resistors and capacitors that has a frequency response similar to the one that our neural network requires. The order of magnitude of the value of the components shows that this circuit could easily be implemented.

By modifying the values of the two resistors within the same order of magnitude as the one shown in Figure 3.3 we could obtain any of the weight functions that our neural network returns.

Until now, everything works. The problem comes with the transformation that we make on the gain. The output gain of the LTI ODE operator is not directly the weight value that we apply during the forward propagation. In order to have stable transfer

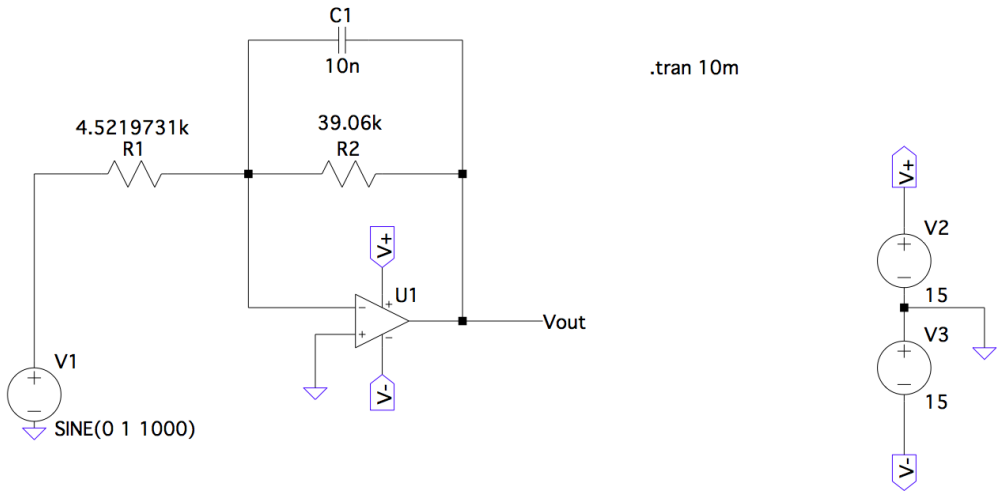


Figure 3.3: Electrical circuit of the weight function.

functions we limited the gain values to 2 ranges and performed a sinus operation on the gain. This sinus operation is really difficult to turn into a electrical circuit. That's why this option is also too complicated to physically implement it.

We have practically exhausted our options. On the first hand we need to limit the possible values of the gain desired at each frequency to achieve stability. But in the other hand, if we restrict the possible values for the gains we cannot physically implement it. In the following chapter a solution to this situation will be introduced.

Chapter 4

Static LTI-ODE operators

As we have seen in the previous chapter it is very difficult to find a LTI ODE operator that is stable and returns the desired gain at each given frequency.

The next approach that has been tried is to train both, AND and XOR neural networks separately using the classical model and obtain the trained weight and bias parameters, which can be identified as the amplitude of the sinus signals of the LTI ODEVNN. Once we have the desired amplitude for each sinusoidal wave we can design a phase controller in the frequency domain with gain 1. In order to implement negative weight values, we will add 180^o to the phase at the desired frequency.

However, if we allow the gain of the LTI ODE operator to take any needed value we are back at square one with the problem of stability. With this approach, as we have seen, it is impossible to guarantee that all the LTI ODE operators will be stable. So we can't find a generic solution for the LTI ODE operator. But maybe, it is possible to find a set of trained weight and bias parameters which can be obtained through a stable LTI ODE operator specifically designed.

Let's think how the Bode plot of the LTI ODE operator should look like in order to increase the chance of it being stable.

First of all, as I have said before, for the LTI ODE operator to be causal, the denominator's degree needs to be higher or at least the same than the degree of the numerator. A pole in the denominator adds $-20db/decade$ and a zero in the numerator adds $+20db/decade$ ([Villa, 2018]). This means that the tendency of the magnitude as we move to higher frequencies is to decrease. In other words, at infinite frequency the magnitude is $-\infty$ dB in the case of a higher degree polynomial in the denominator and a static value in the case of the same degree in both the numerator and denominator. This means that we should find a solution for the neural network in which the gain at the highest frequency (1000 Hz in our case) is smaller than the gain at the lowest frequency. If this is not possible, we can have greater gains at the highest frequency if we increase the degree of the denominator. But this would quickly complicate the task of finding a stable transfer function.

The other issue that needs to be taken into account is that even if we find a set of weights in which the gain at the highest frequency is smaller than the gain at the lowest frequency, if the values of the two gains are very different, the frequency response of the function will have to change a lot in a small frequency window. To achive such a jump in the magnitude response the degree of both the numerator and denominator would also increase.

To understand this I will propose an example. In our LTI ODEVNN, the two problems that we are trying to solve are codified by the following frequencies: 100 Hz and 1000 Hz. The distance between these two frequencies in a logarithmic scale is one decade. As I have previously said, a pole in the denominator adds $-20db/decade$ and a zero in the numerator adds $+20db/decade$. This means that if we increase by one the degree of the numerator or denominator we can achieve maximum a gain jump of 20 dB.

So, in order to keep our LTI ODE operator simple, we also want the weight values at each frequency to be similar. This way, the gain jump that we need to do will be smaller.

From what we have seen so far, it doesn't seem an easy task either. What was proposed to do was to train the neural network a lot of times until we found the set of weights that were closest to the requirements mentioned above. To do so, we used the `fminsearch` function of Matlab, which uses the gradient descent to minimize a function.

The parameter to minimize is: for each connection –in our LTI ODEVNN there are 6 connections between neurons– calculate the squared difference between the gain at 100 Hz and the gain at 1000 Hz and perform the sum of the calculated values for the 6 connections. To minimize this parameter represents bringing closer the values of the gain at 100 Hz and the gain at 1000 Hz. As I have just explained this is important to keep the LTI ODE operator simple and to have a chance at it being stable.

The minimization function that we implemented, at each step of the iterative process makes adjustments to the initial weights and biases and then trains the neural network. Once the neural network is trained, checks whether or not the training process has converged. In other words, it checks if it has been found a set of weights and biases that solve the AND and XOR problems. If the LTI ODEVNN has been successfully trained, evaluates the parameter to be minimized and through gradient descent makes the necessary adjustments. And the process repeats itself all over again.

What we found by doing this minimization is that, eventually, the parameter to be minimized becomes nearly 0. This means that exists a set of weights and biases that solve the LTI ODEVNN where the weight values for the AND and XOR problems are the same. This is an amazing result. It means that the same set of weights can solve two different problems, and that the only difference lays in the biases, which are different for each problem. This means that our LTI ODE operators have become the simplest one possible: a static gain constant for all the frequencies. We would only need to focus on the biases, which as we will see in the following chapters, are easy to physically implement.

Approach	Stable	Plausible	Num.	D(s)
Expected LTI ODE	Yes	Yes	3	-
Polynomial	No	No	3	$a_5s^5 + a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0$
Rational polynomial	No	Yes	3	$\frac{b_2s^2+b_1s+b_0}{s^3+a_2s^2+a_1s+a_0}$
1st stable approach	No	Yes	3	$\frac{1}{as^2+bs+c}$
2nd stable approach	Yes	No	2	$\frac{1}{as+b}$
Static LTI ODE	Yes	Yes	3	k

Table 4.1: Evolution of the LTI ODE operator: static LTI ODE operators.

To be sure that this was not a very particular case, I wrote a Matlab code to train a neural network that simultaneously solved two problems (AND and XOR) where the LTI ODE operators at each connection between neurons were a static gain. This means that the amplitude of the sinusoidal waves of 100 Hz and 1000 Hz are amplified by the same value. The weights and biases were also randomly initialized. The results obtained show that what we have found through the minimization procedure wasn't a particular case. There are a lot of possible values for the biases and the static gains that solve the neural network. Later in this chapter, I will make a deeper analysis on the relationship between the initial values of the biases and LTI ODE operators and the final trained values.

After training a neural network that successfully solves two problems simultaneously using static gains as LTI ODE operators, I wondered whether this solution would also work when trying to simultaneously solve 3 problems. I adapted the Matlab code to include a third problem codified in a frequency of 500 Hz. The problem to be solved that I included is the Boolean function OR. It worked well. The algorithm converges and finds a set of biases and LTI ODE operators that solve all 3 problems simultaneously. In Annex B there is the code written in Matlab and Python that is used to train this neural network. In the following section, I will explain in detail how the code works. To make the explanation I will use the Matlab code. The python code works in exactly the same way.

In table 4.1 we can see that by using static LTI ODE operators we have finally achieved the expected results: stability, physical plausability and the possibility of simultaneously solving 3 problems.

4.1 Static gains based neural network: code

In this section I will explain in detail how the code for training a neural network that simultaneously solves 3 problems works. At the end of the section I will present a set of static gains and biases that solve the neural network. The neural network that we are building tries to simultaneously solve the following 3 problems: the logic gates AND, XOR

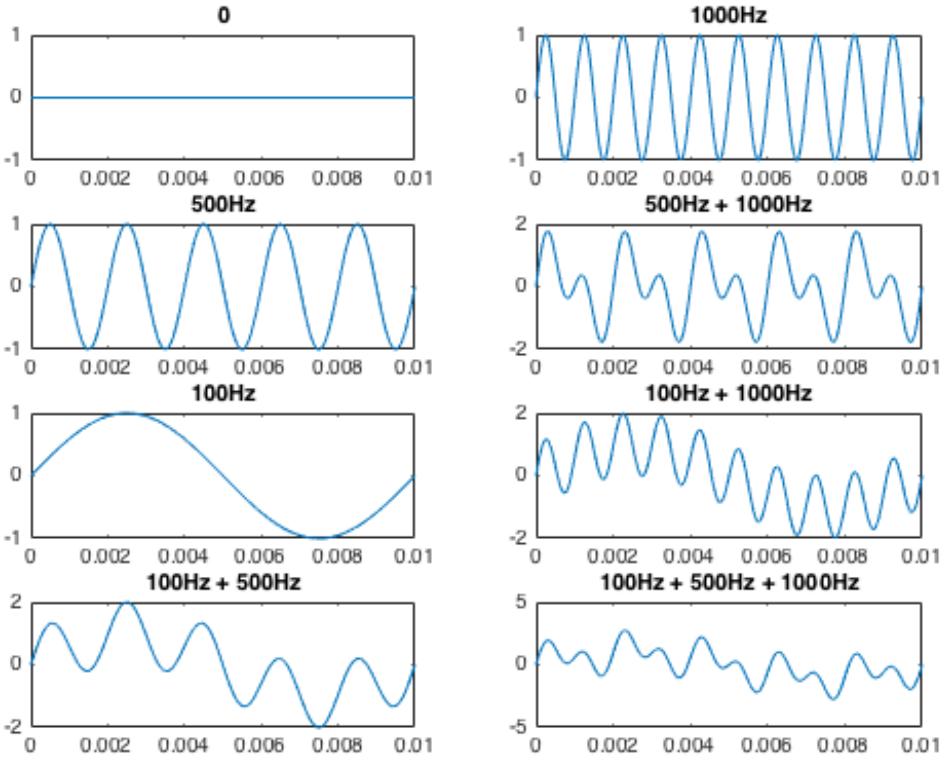


Figure 4.1: Possible inputs of the neural network.

and OR. The signal that goes through the neural network is a sum of three sinusoidals with 3 different frequencies. Each sinusoidal indenfies one problem:

- Logical gate OR: sinuosoidal wave of frequency 100 Hz.
- Logical gate AND: sinusoidal wave of frequency 500 Hz.
- Logical gate XOR: sinusoidal wave of frequency 1000 Hz.

Let's talk now about the different scripts that conform our LTI ODEVNN:

- `training-data.m`: in this file I generate the inputs of the neural network and the expected outputs of the neural network for each combination of inputs. Our neural network has two inputs. Each input is a sum of three sinusoidal waves of the frequencies mentioned above. The amplitude of the input sinusoidals is 1 or 0. In a electrical circuit, a 1 can be identified for example with 5 volts. In Figure 4.1, we can see all the possible combinations of sinusoidal waves that we can have at the input signal. The expected outputs of the neural network are the result of performing the logical operations OR, AND and XOR between the two inputs.
- `main.m`: in this file I first introduce the training data to be used to train the LTI ODEVNN. Afterwards, I randomly initialize the values of the biases of each neuron

and the static gains for every connection between neurons. Then, the iterative learning process begins. This process has two stages: forward propagation and back propagation.

- `forwardpropagation.m`: this file performs the weighed sum of the inputs for each neuron and calculates the final output of the neural network.

- `activation.m`: this file is a Matlab function that takes as argument the weighted sum of the inputs to a neuron (z) and returns the output of the neuron. The activation function used is the sigmoidal function $\frac{1}{1+e^{-z}}$.

- `backpropagation.m`: this file calculates the derivatives of the error for each weight and bias and makes the corresponding adjustments. It returns the new static gains and the new biases.

All the files are important, but the last one is the most complex. The calculation of the derivatives is made according to the BackPropagation algorithm explained in Section 2.3. What makes it tricky to calculate the derivatives is the fact that our weight function is a single value for the 3 problems. And when we calculate the derivatives of the error each problem has a different derivative for the same weight. What that means is that the direction in which for example the XOR problem minimizes the error may not be the same direction as the one that the AND problem requires. This wouldn't be a problem if each problem had its own set of weights. But in our case they share the same weight. So we have to find a way of doing that. Now I will explain how I have approached this task.

First of all, it's important to take into account that as we have seen, the importance of the learning process lays more in the biases than in the weights themselves. This means that we can sum the derivatives with respect to the weight for all three problems. The weights will then be optimized according to the 3 problems at the same time. Following this method, the training process converges and finds a solution, but not always. Sometimes it gets stuck in a set of weights where the derivatives are close to 0 and the learning process stops. This didn't happen as often when we were training a neural network with 2 problems. The reason for that is that probably when incorporating a 3rd problem to the neural network, the function to be minimized becomes more complex.

To improve the success training rate I tried two methods: incorporating a momentum parameter to the training process and the introduction of a new parameter called complexity.

4.1.1 Momentum parameter

The momentum is a way of avoiding local minima that prevent the convergence of the training process. As we have seen in Section 2.3, the formula to update the weights and

biases is : $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\Delta E(\mathbf{w}^{(\tau)})$. And at each iterative step, the derivative of the error changes. When we introduce the momentum into the equation, what we do is to update the weights and biases using the derivative of the error calculated at the current iteration plus the sum of all the previous adjustments multiplied by a parameter called momentum ([Duda et al., 2000]).

For example, let's imagine that we are training a neural network. At the first iteration, we compute the derivative of the error for a given weight and multiply it by the learning rate according to the formula. This value, that we will call $\mathbf{v}_0$ is the adjustment to be made to the weight. Then, at the second iteration, we will compute the new adjustment $\mathbf{v}_1$. But now, we will update the weight following the following formula: $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - (\mathbf{v}_1 + \alpha\mathbf{v}_0)$. So, the general formula for the momentum is for iteration n:

$$\mathbf{w}^{(n)} = \mathbf{w}^{(n-1)} - (\alpha\mathbf{v}_{n-1} + \alpha^2\mathbf{v}_{n-2} + \ldots + \alpha^{(n-1)}\mathbf{v}_1 + \alpha^n\mathbf{v}_0) \tag{4.1}$$

The momentum parameter α has a value between 0 and 1. This means that the adjustments from the most recent iterations are more relevant than the adjustments from the oldest iterations, because if we raise to a power a value between 0 and 1 it decreases exponentially as the power increases.

At the beginning of the training process, normally the value of the derivatives of the error are large. This means that at the first iterations, the effect of the momentum will be higher that when we are close to the global minima and the derivatives are smaller. The momentum helps avoiding local minima because of the 'intertia' that it provides.

The downside to using momentum is that the value of the parameter isn't trained during the training process and you have to use trial and error to find a good value. In my case I have tried different values and I have finally set for a momentum parameter of 0.1. Anyway, including a momentum to my neural network hasn't significantly improved the success rate.

4.1.2 Complexity parameter

After incorporating the momentum parameter into the training process and realizing that the results haven't significantly improved I incorporated the complexity parameter. As I have explained before, to update the values of the weights and biases I performed the sum of the derivatives from the 3 problems so at the end each problem equally contributed to the adjustment made. But maybe the contribution to the update should be different for each problem to obtain better results. What I observed from the times when the training process didn't find a solution is that the problem that failed to converge was the Boolean function XOR. The problems AND and OR absolutely always converged. This can be explained because the logical XOR is a more complex problem than the other two problems: the outputs of the Boolean function XOR can't be linearly separated by a

single line. This is why we need 3 neurons in the first place; to solve the XOR problem.

So that being said, the complexity parameter is easy to understand. What it does is to give more relevance to the derivative of the error that comes from the XOR problem so the weights get updated more accordingly to this one problem. The complexity parameter that I use to train the neural network is: 0.1 for the AND and OR problems and 2 for the XOR problem. These values work fine, but there are others that could work. The key factor here is to give more importance to the XOR problem. After incorporating the complexity parameter the success rate of trained neural networks increased significantly.

Finally there is another parameter to take into account, the learning rate. The learning rate is a parameter that indicates how big is the step that is going to be made in the direction of minimization of the error function. If you pick a small learning rate, the training process will proceed slowly but surely. Conversely, if you pick a large learning rate, you could overshoot a good answer and then on the next iteration undershoot, and get into an oscillating pattern where training never converges. The value of the learning rate used in the training process has also been found by trial and error and is 0.7. I have found that this value meets a good compromise between quick learning and security.

4.1.3 Results

Now that we have a deep understanding of the code that trains the LTI ODEVNN, I will present the results and one of the sets of weights and biases that solve the neural network.

```
r =
         0         0    0.0295    0.0216    0.0325
         0    1.0000    0.9715    0.0157    0.9600
    1.0000         0    0.9708    0.0180    0.9598
    1.0000    1.0000    0.9771    0.9514    0.0597
```

In the results showed above, the first two columns are the two inputs and the next 3 columns are the output of the trained neural network for each problem.

Now let's see which is the set of static gains and biases that are used to obtain these results. To make it more understandable, the following matrix shows how the values are distributed:

$$\begin{bmatrix} neuron1: & bias_{OR} & bias_{AND} & bias_{XOR} & weight_{input1} & weight_{input2} \\ neuron2: & bias_{OR} & bias_{AND} & bias_{XOR} & weight_{input1} & weight_{input2} \\ neuron3: & bias_{OR} & bias_{AND} & bias_{XOR} & weight_{input1} & weight_{input2} \end{bmatrix}$$

```
w =
   -1.5103   -0.8544    5.1590   -3.7071   -3.7116
    2.6693    7.3201    2.0122   -6.0941   -6.1781
    3.7549    3.0373   -3.5602    8.5007   -9.3942
```

In the following section I will do a deep analysis about the evolution of the weights and biases.

4.2 Analysis of the evolution of the weight and bias parameters

Now that we have a way of successfully training a LTI ODEVNN that simultaneously solves 3 problems, we should analyise the evolution of the weight and bias parameters as we train the neural network. Do the random initialization of the paramenters affects the final convergence of the training process or not? Are the set of weights and biases that solve the neural network always similar or do they vary a lot? In this section these questions will be answered.

The first analysis consists in training the LTI ODEVNN with a large number of iterations to see how the weights and biases evolve during the training process. What it has been done is to train the neural network with 30000 iterations. In Figure 4.2 and Figure 4.3 we can observe the results of the analysis.

In Figure 4.2 there's the analysis for the biases separated for each problem. We can see that the value of the biases converges at the beginning of the training process. At the first 100 iterations the value of the biases changes quickly. After this initial stage, the value of the biases stabilizes really quickly and doesn't change much during the rest of the training process. This can help us choose an appropriate number of iterations for the training process. It can be seen that the learning process of the biases takes the same amount of time for the hidden layer as well as for the output layer of neurons.

Let's analyse now the evolution of the LTI ODE operators (static gains). We have 3 neurons organized in two layers. This makes 6 connections between neurons, and since we have the same weight value for all three problems, in total we have 6 weights. The evolution of these 6 values is showed in Figure 4.3. It happens the same as with the biases. The weights converge at the first iterations very quickly.

What may seem odd when observing the plot is that instead of 6 weights it seems that we have only 4 weights. That is because the weights from the hidden layer neurons converge to the same value for both the inputs of the neuron. As we will see in more detail in the next analysis, most of the times that we train the neural network the set of weights present this behaviour. If we zoom at the first 100 iterations, it can be observed that we have 6 different lines corresponding to the 6 weights. During the initial iterations

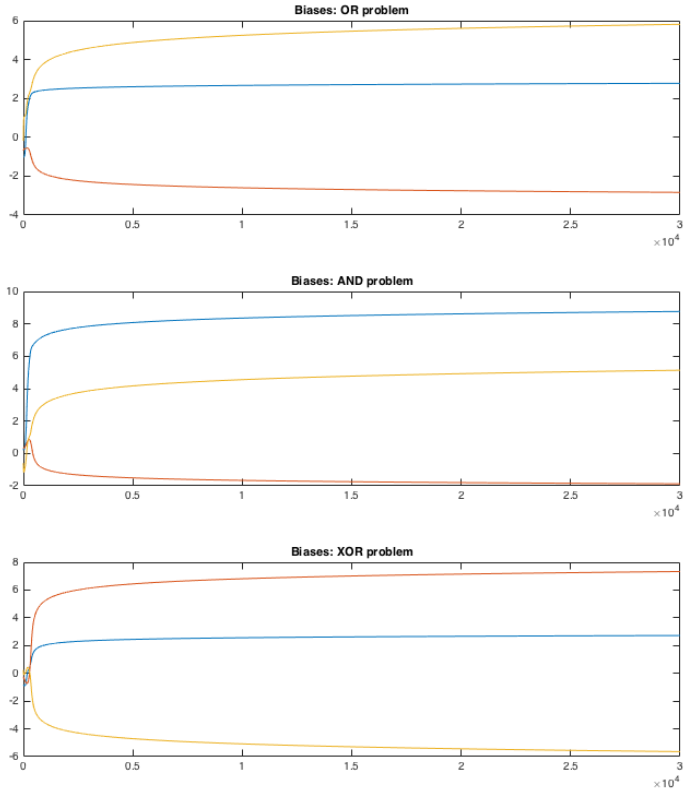


Figure 4.2: Evolution of the biases. There are 3 lines at each plot. Each line corresponds to the bias of one neuron.

the weights from the hidden layer still hadn't had time to converge to the same value.

The next analysis consists in the relationship between the initial random values and the final learned values. What has been done is to train the LTI ODEVNN a thousand times. Each time with a different set of initial weights and biases, the values are choosen uniformly from the interval $[-1, 1]$. In Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.8, Figure 4.9 and Figure 4.10 the results of this analysis are shown.

In the plots there are dots in 4 different colours. Blue and magenta colour means initial values. Red and green colours are identified with the final learned values. The plots also differentiate the set of weights and biases that solve the neural network for all 3 problems (good convergence) from the set of values that don't solve all three problems (bad convergence). So, colours blue and green mean good convergence; and magenta and red bad convergence.

Let's analyse the plots:

- The first thing that can be observed is that there is no difference between the initial values that lead to a good convergence than the ones that lead to a bad convergence. They are equally distributed inside the one-by-one square at the center of each plot. In Figure 4.7 I have made a zoom at the initial square so it can be observed that

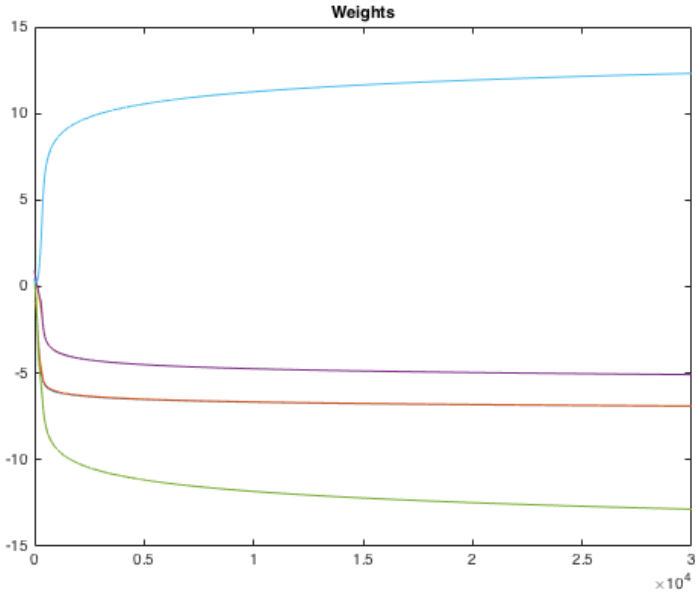


Figure 4.3: Evolution of the weights.

there is no pattern that indicates that some initial values are better that some other ones.

- From the plots we can also see that the final values concentrate in certain areas of the plots.

- In the plots of the weights from neuron 1 and neuron 2 we can see that most of times the weights from both inputs take the same value. But not always. Some rare times, as we can see in the plots, they take the same absolute vale but with different sign.

- From the plots it seems that there are more red dots that green dots. But that is because the areas where the green dots concentrate are more dense. The rate of success of training can also be calculated from this analysis:

$$\text{success rate} = \frac{\text{Successed trainings}}{\text{Failed trainings}} = \frac{526}{1000} = 0.526 \tag{4.2}$$

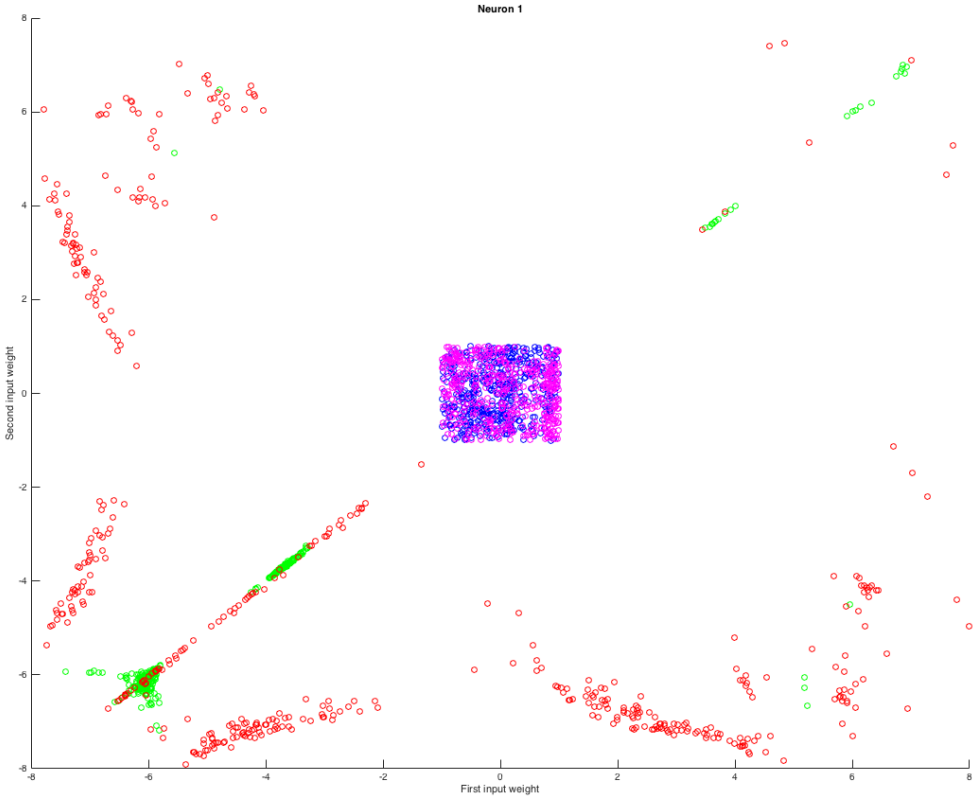


Figure 4.4: Initial vs final weights: Neuron 1.

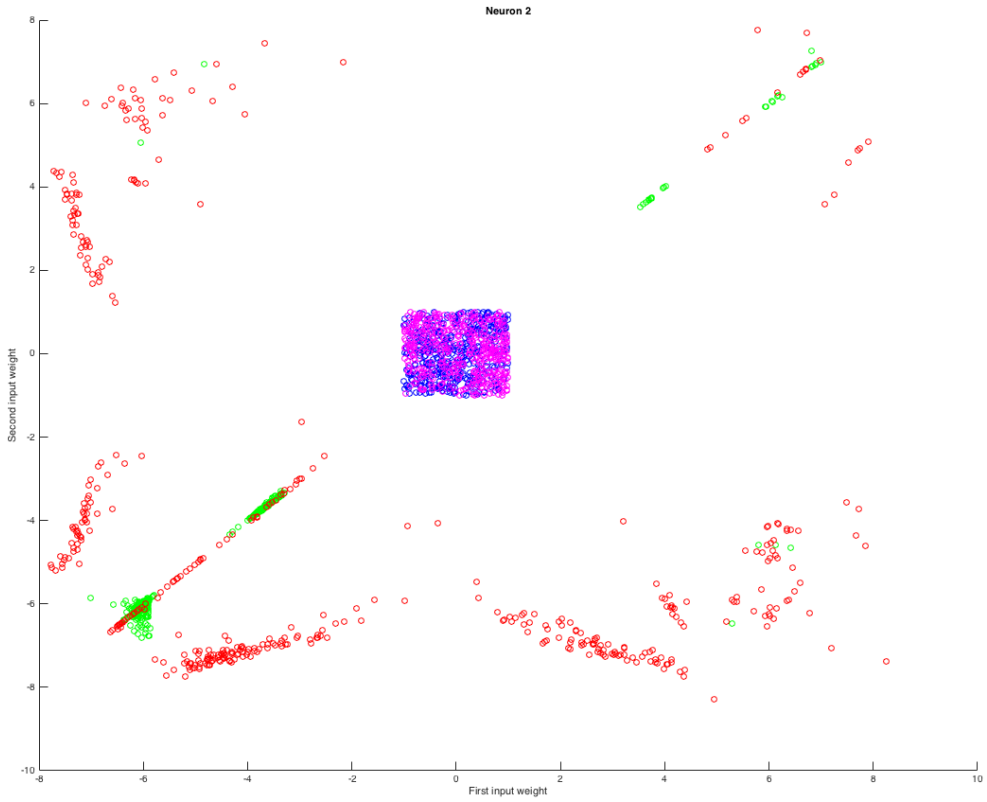


Figure 4.5: Initial vs final weights: Neuron 2.

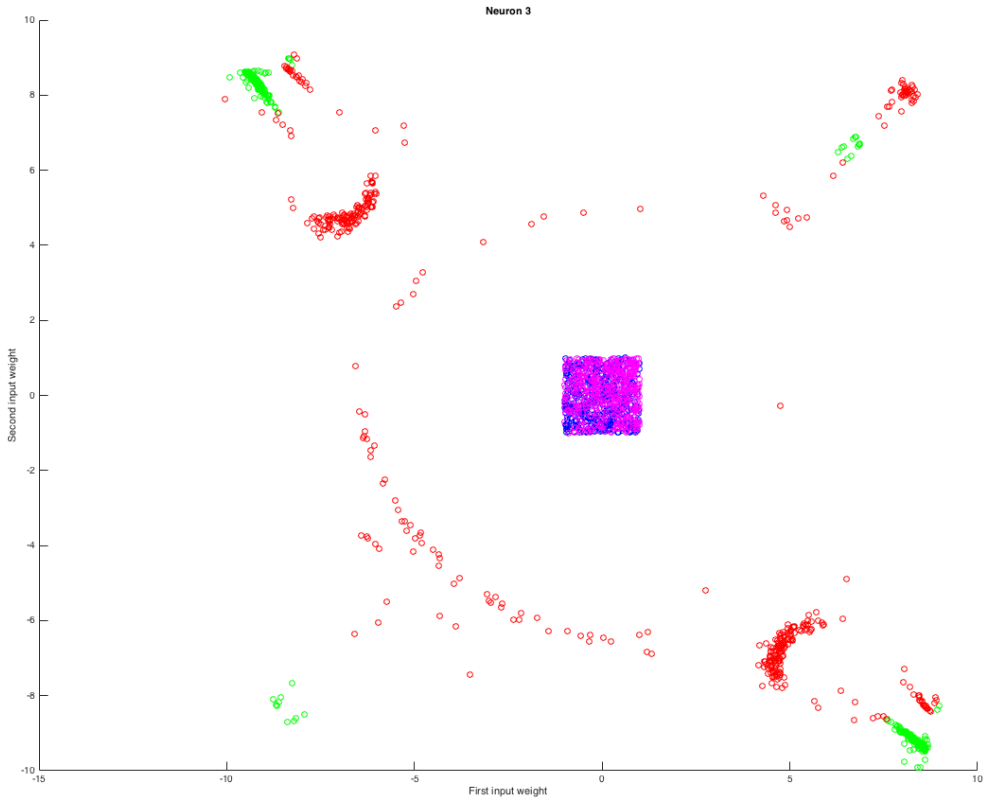


Figure 4.6: Initial vs final weights: Neuron 3.

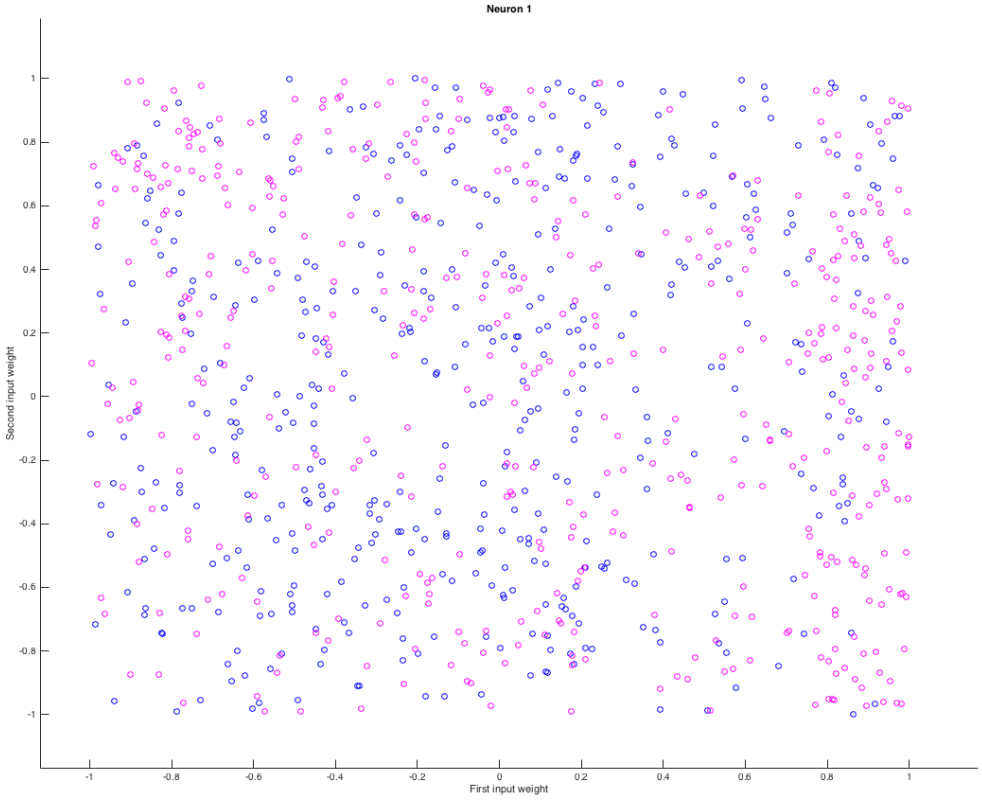


Figure 4.7: Zoom at the initial square of values.

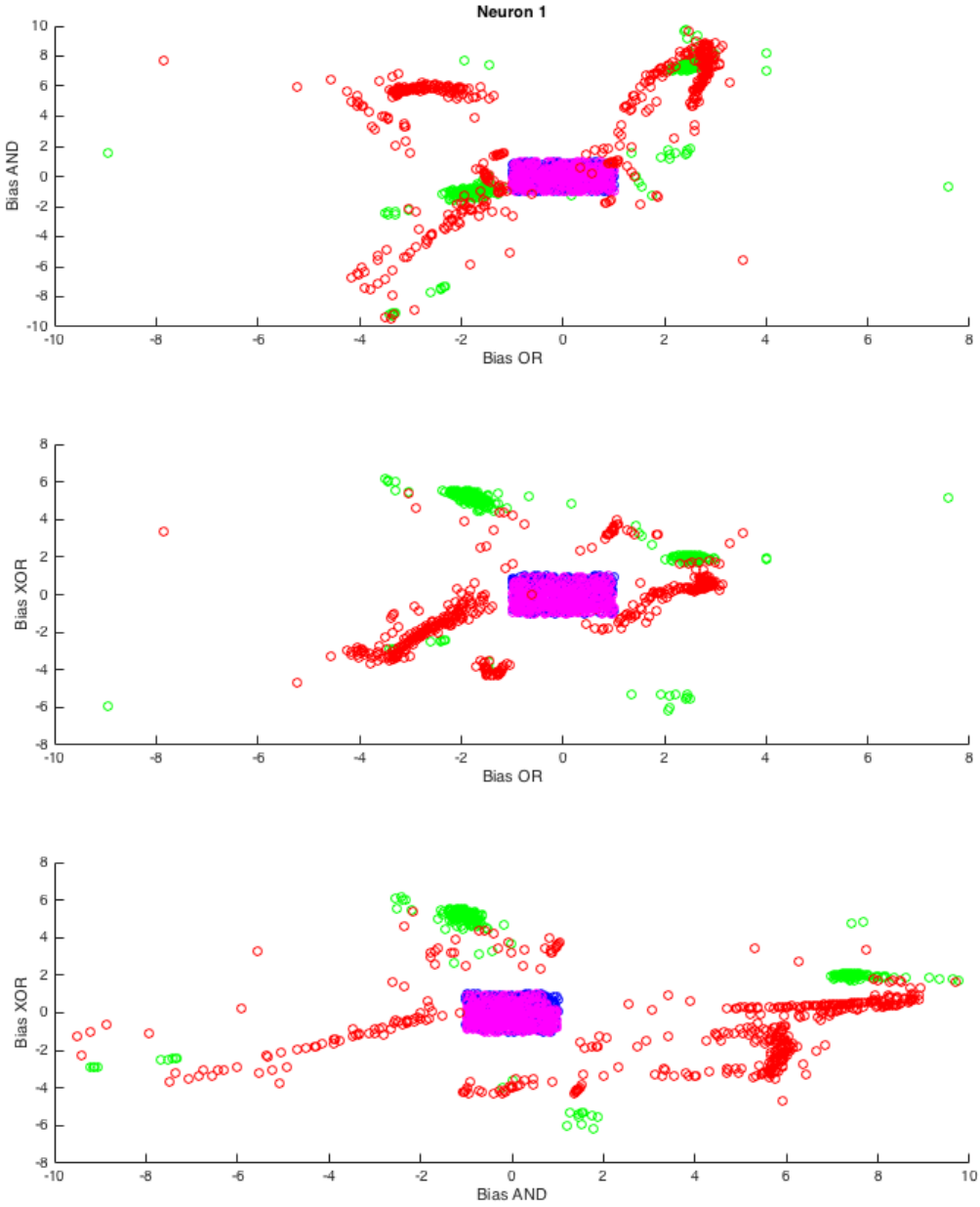


Figure 4.8: Initial vs final biases: Neuron 1.

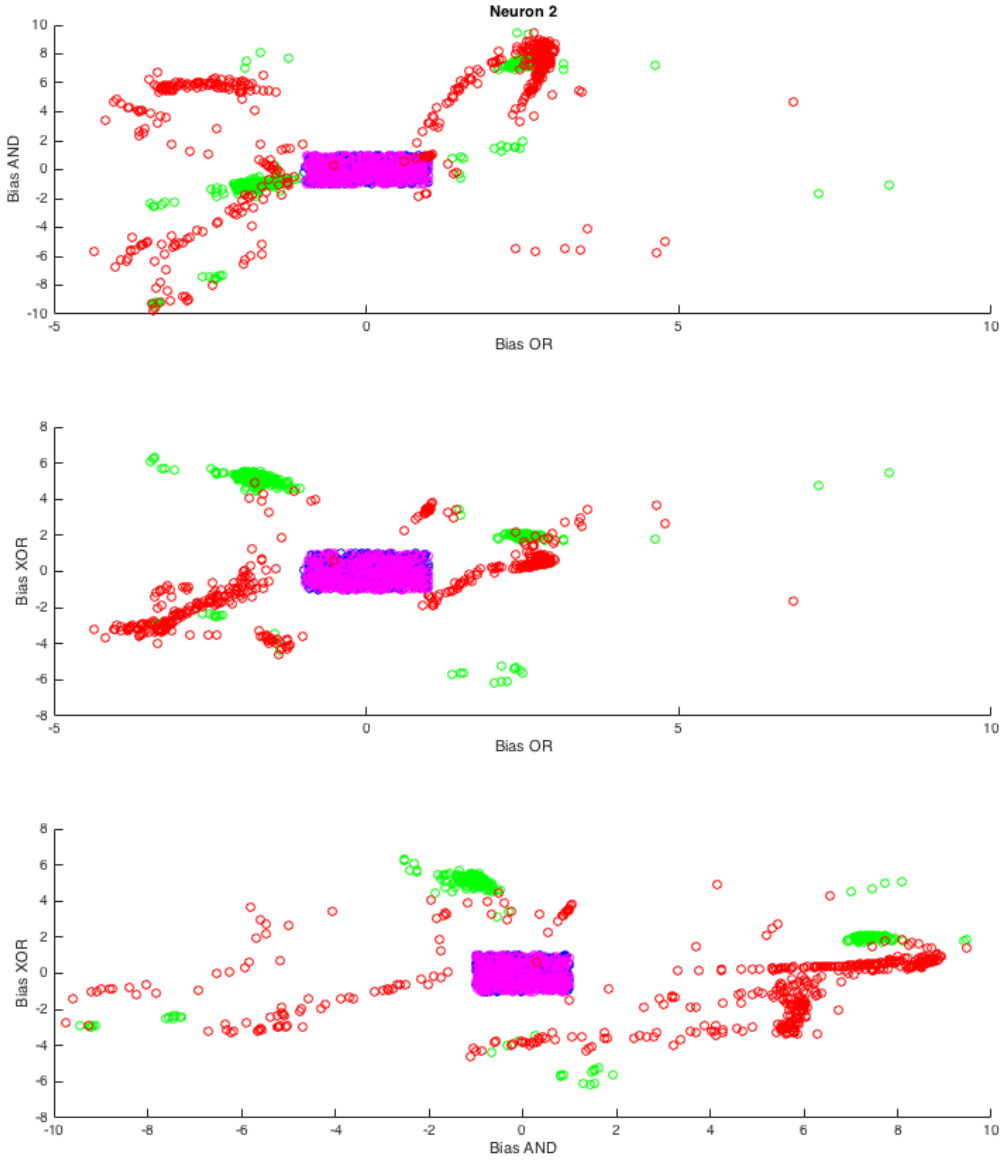


Figure 4.9: Initial vs final biases: Neuron 2.

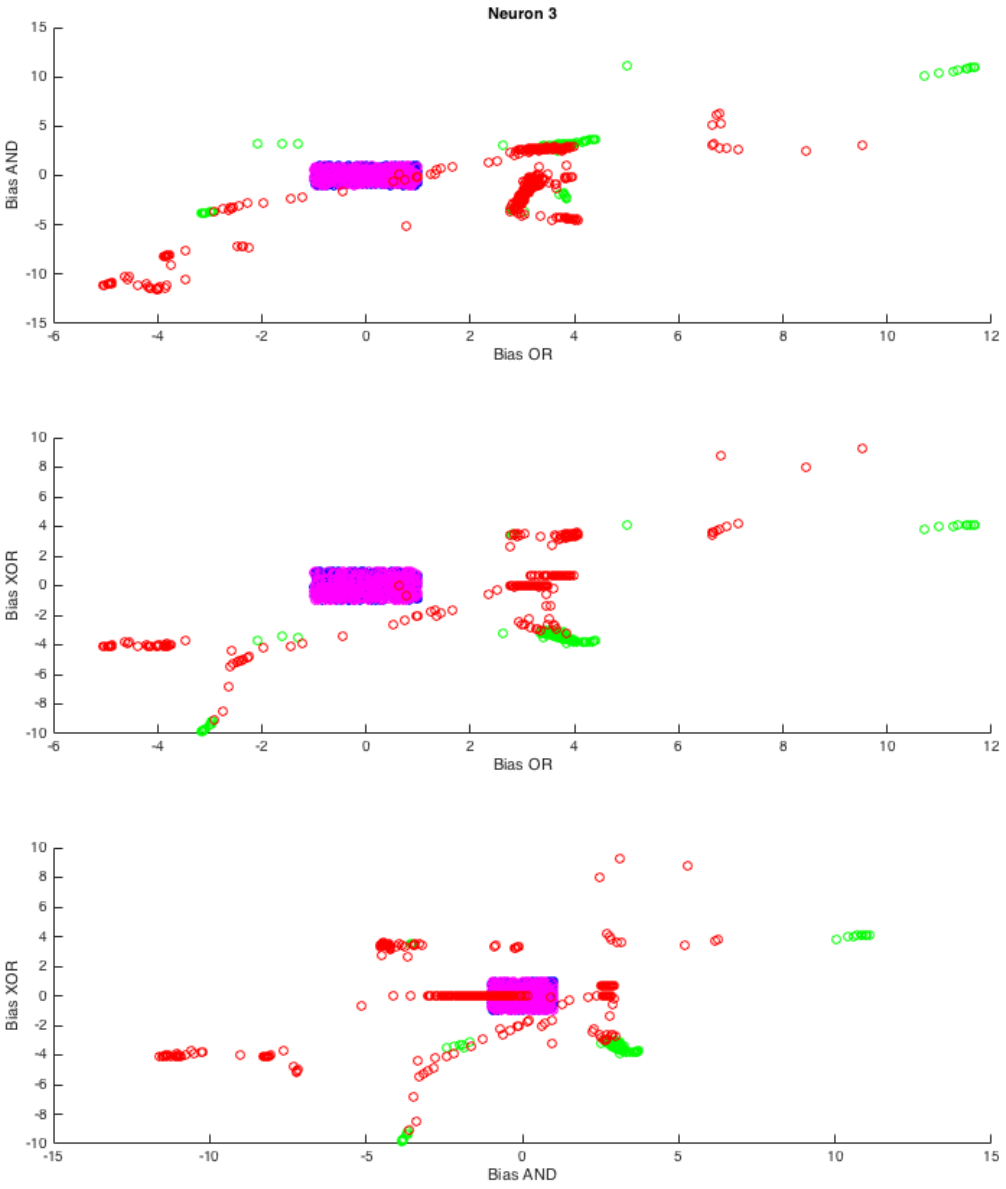


Figure 4.10: Initial vs final biases: Neuron 3.

Chapter 5

Costs

This project has been completed in a period of 20 weeks. In Figure 5.1 we can see the Gantt diagram of the project.

- Analysis of the article 'LTI ODE-valued neural networks' [Velasco et al., 2014]: 3 weeks. During this period I studied the article in order to reach a deep understanding of the project and to start the project in the appropriate direction.
- Study of classical neural models: 3 weeks. Using external bibliography from books and Internet I studied the classical model of feed-forward neural network and the BackPropagation algorithm through gradient descent optimization.
- Development of the LTI ODE operator: 10 weeks. This is the main part of the project and the one in which I have invested more time.
- SPICE simulation: the implementation of the LTI ODEVNN into an electrical circuit is the last part of the project. It is still a work in progress.

The writing of the memory has been a process mainly done during the last stages of the project. But also during the development of the LTI ODE operators I have been keeping record of the progress by writing isolated parts of the memory.

To calculate the cost of the project I will take into account the cost of the working personal (one person).

The whole project has taken 20 weeks, with 25 hours average per week. I we consider a cost/hour of 15€/h, then the final cost of the project is: 7500€.

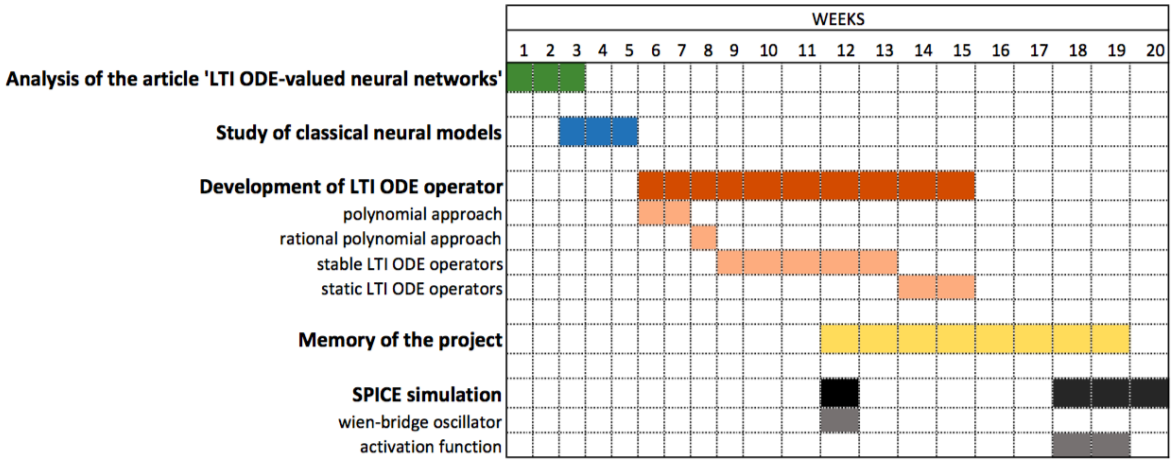


Figure 5.1: Gantt diagram. Week 1 corresponds to the first week of February, 2018. Week 20 corresponds to the last week of June.

Environmental impact

This project is virtually entirely a theoritical development of a LTI ODEVNN (linear time-invariant ordinary differential equations valued neural network). The only resource used to study these kind of neural networks has been a computer, which utilizes electricity as a power supply. The supply of electricity of a computer is very small (at least in the case of the computer used for the develpment of this project) , but any consumption of electricity has an environmental impact.

Moreover, on the second part ot the project, we have designed the LTI ODEVNN in the form of an electrical circuit using a SPICE simulator. The circuit hasn't been physically implement during the course of this project, but it is the purpose ot this work to develop a model of LTI ODEVNN that can be implemented into an electrical circuit. So it makes sense to take into account that the final results of this project can be physically implemented in the form of an electrical circuit, which is formed by resistors, capacitors, operational amplifiers, cables, etc. The process of fabrication of all this components has an environmental impact and also the process of assembling the circuit, if automatized in a industrial process, has an impact to the environment.

Conclusions

At the end of the project it has been successfully trained a LTI ODEVNN that simultaneously solves 3 problems: the Boolean functions AND, XOR and OR. The final form of the LTI ODE operators is not what was proposed at the beginning. At the end we have used LTI ODEs with static gain because we have learned that what matters most in the LTI ODEVNN are the bias parameters instead of the weights. Also, the quest of finding stable LTI ODE operators has been very difficult.

In regard with the SPICE simulation, it is a part of the project that hasn't been accomplished due to a lack of time. The design of an analogical circuit for the activation function of the neuron has been more complicated than anticipated. I will keep working on this subject.

During the development of the project, some of the open questions left in the article [Velasco et al., 2014] have been answered:

- The first question was the relantionship between the order of the employed ODE and the maximum number of solvable problems for the LTI ODEVNN. We have seen that as we increase the order of the LTI ODE operators, the stability conditions become bery restrictive and complicated. We have had to settle for the most simple form of LTI ODE because of that. So probably, the way of increasing the number of solvable problems is not to increase the order of the employed LTI ODEs.

- The second question was about the possibility of reordering the problems (the frequencies associated to each problem) to minimize the complexity of the neural network. The final form of the LTI ODE operators that we have designed is not affected by the order of the frequencies, since the values of the bias and weight parameters for each problem are trained separately.

- The last question posed was whether the system's stability could be guaranteed, as well as the convergence to solutions. We have seen that a part from an LTI ODE operator with static gain for every frequency, it has been very difficult, if not impossible, to find stable LTI ODEs which accomplish their function of controlling the amplitude of the sinus signals. That is a question that remains open. Maybe there are other ways, or slightly different models in which stable LTI ODEs can be found.

Regarding the question of convergence to solutions, in the analysis of the results we have seen that not always the training process converges. But more than 50% of the times the process converges. We have also seen that the convergence to a solution does not depend on the random initial values of the bias and weight parameters.

It is left for future studies to try to prove if there is a general method that can guarantee if a LTI ODEVNN will or not converge to a solution.

This project is only a starting point to the study of LTI ODEVNNs. There is still a lot of work to do. We have used a simple toy example to simulate and train our model. The next step should be to study if static LTI ODE operators are still adequate for more complex problems.

Another question open for future studies is whether there is a limit in the number of problems that can be solved by a LTI ODEVNN.

Bibliography

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[Duda et al., 2000] Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification (2Nd Edition)*. Wiley-Interscience, New York, NY, USA.

[Sardi et al., 2017] Sardi, S., Vardi, R., Sheinin, A., Goldental, A., and Kanter, I. (2017). New types of experiments reveal that a neuron functions as multiple independent threshold units. *Nature. Scientific Reports*.

[Velasco et al., 2014] Velasco, M., Martín, E. X., Angulo, C., and Martí, P. (2014). LTI ODE-valued neural networks. *Applied Intelligence*, 41(2):594–605.

[Villa, 2018] Villa, R. (2018). *Dinamica de sistemes*. ETSEIB, Barcelona, Spain.

Appendix A

Matlab code

A.1 Polynomial approach

```
% Definicio de variables
syms s a b c d e f real

% Definicio de les frequencies de treball i dels input
W = [10 50 95];
x1 = W(1);
x2 = W(2);
x3 = W(3);

x = 1;

% Definicio del guany objectiu a les frequencies de treball
T = [90 + 2i, -20 - 7i, 50 + 10i];

% Definicio dels punts inicials
Y = [-10 + 89i,-8-10i,10+1i];

% Numero de punts
n = length(W);

% Learning rates
lr = 0.1;

% NÃžmero d'iteracions
num = 50;

% CÃ¤lcul del polinomi objectiu
eqs = [];
variables = [a b c d e f];
```

```
30
31 for j = 1:3;
32     eq1 = b * W(j)^4 - d * W(j)^2 + f == real(T(j));
33     eq2 = a * W(j)^5 - c * W(j)^3 + e * W(j) == imag(T(j));
34     eqs = [eqs eq1 eq2];
35 end
36
37 [a b c d e f] = solve(eqs,variables);
38
39 p_coef_0 = double([a b c d e f])
40 obj = a * s^5 + b * s^4 + c * s^3 + d * s^2 + e * s + f;
41
42 % CÃălcul del polinomi
43 syms a b c d e f real
44 eqs = [];
45 variables = [a b c d e f];
46
47 for j = 1:3;
48     eq1 = b * W(j)^4 - d * W(j)^2 + f == real(Y(j));
49     eq2 = a * W(j)^5 - c * W(j)^3 + e * W(j) == imag(Y(j));
50     eqs = [eqs eq1 eq2];
51 end
52
53 [a b c d e f] = solve(eqs,variables);
54
55 p_coef = double([a b c d e f]);
56
57 % plot resultats
58 xx = linspace(0,100);
59 yy = polyval(p_coef_0,xx);
60 plot(xx,yy,'blue')
61 hold on
62 yy = polyval(p_coef,xx);
63 plot(xx,yy,'green')
64
65 % PROCES ITERATIU
66
67 a = p_coef(1);
68 b = p_coef(2);
69 c = p_coef(3);
70 d = p_coef(4);
71 e = p_coef(5);
72 f = p_coef(6);
73
74 da = [ -(- x2^3*x3 + x2*x3^3)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1^2*
       x3^2 - x2^2*x3^2 + x3^4)), (- x1^3*x3 + x1*x3^3)/(x1*x2*x3*(x1^2 - x2
       ^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4)), -(- x1^3*x2 + x1*x2
```

```
      ^3)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3
      ^4))];
75 db = [ (x2^2 - x3^2)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 +
      x3^4)), -(x1^2 - x3^2)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*
      x3^2 + x3^4)), 1/(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4)];
76 dc = [ -(- x2^5*x3 + x2*x3^5)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1^2*
      x3^2 - x2^2*x3^2 + x3^4)), (- x1^5*x3 + x1*x3^5)/(x1*x2*x3*(x1^2 - x2
      ^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4)), -(- x1^5*x2 + x1*x2
      ^5)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3
      ^4))];
77 dd = [ (x2^4 - x3^4)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 +
      x3^4)), -(x1^4 - x3^4)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*
      x3^2 + x3^4)), (x1^4 - x2^4)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 -
      x2^2*x3^2 + x3^4))];
78 de = [ -(- x2^5*x3^3 + x2^3*x3^5)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1
      ^2*x3^2 - x2^2*x3^2 + x3^4)), (- x1^5*x3^3 + x1^3*x3^5)/(x1*x2*x3*(x1
      ^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4)), -(- x1^5*x2^3
       + x1^3*x2^5)/(x1*x2*x3*(x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*
      x3^2 + x3^4))];
79 df = [ -(- x2^4*x3^2 + x2^2*x3^4)/((x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 -
       x2^2*x3^2 + x3^4)), (- x1^4*x3^2 + x1^2*x3^4)/((x1^2 - x2^2)*(x1^2*
      x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4)), -(- x1^4*x2^2 + x1^2*x2^4)/((
      x1^2 - x2^2)*(x1^2*x2^2 - x1^2*x3^2 - x2^2*x3^2 + x3^4))];
80
81
82 for j=1:num
83
84 p = a * s^5 + b * s^4 + c * s^3 + d * s^2 + e * s + f;
85 suma = p * x;
86 z = suma;
87 error = z - obj;
88
89 % Vectors error
90 E_real = real(double(subs(error,i*W)));
91 E_complex = imag(double(subs(error,i*W)));
92
93 a = a - lr * da * transpose(E_complex);
94 b = b - lr * db * transpose(E_real);
95 c = c - lr * dc * transpose(E_complex);
96 d = d - lr * dd * transpose(E_real);
97 e = e - lr * de * transpose(E_complex);
98 f = f - lr * df * transpose(E_real);
99
100 p_coef = double([a b c d e f]);
101
102 % Plot resultats
103 yy = polyval(p_coef,xx);
```

```
104 plot(xx,yy,'red')
105
106 end
107
108 p_coef
```

A.2 Rational polynomial approach

```
% Definicio de variables
syms s a b c d e f real

% Definicio de les frequencies de treball i dels input
W = [10 50 95];
x1 = W(1);
x2 = W(2);
x3 = W(3);

x = 1;

% Definicio del guany objectiu a les frequencies de treball
T = [90 + 10i, -20 - 7i, 50 + 10i];

% Definicio dels punts inicials
Y = [-10 + 89i,-8-10i,10+1i];

% Numero de punts
n = length(W);

% Learning rates
lr = 0.1;

% Numero d'iteracions
num = 100;

% Calcul del quocient de polinomis objectiu
eqs = [];
variables = [a b c d e f];

for j = 1:3;
    r = real(T(j));
    cx = imag(T(j));

    eq1 = -r*W(j)^3 + r*e*W(j) - cx*d*W(j)^2 + cx*f == b*W(j);
    eq2 = -r*d*W(j)^2 + r*f + cx*W(j)^3 - cx*e*W(j) == -a*W(j)^2 + c;
    eqs = [eqs eq1 eq2];
end

[a b c d e f] = solve(eqs,variables);

n_coef_0 = [a b c];
d_coef_0 = [1 d e f];
```

```
n_0 = a*s^2 + b*s + c;
d_0 = s^3 + d*s^2 + e*s + f;

obj = n_0/d_0;
coef_obj = double([a b c d e f])

% Calcul del quocient de polinomis
syms a b c d e f real
eqs = [];
variables = [a b c d e f];

for j = 1:3;
    r = real(Y(j));
    cx = imag(Y(j));

    eq1 = -r*W(j)^3 + r*e*W(j) - cx*d*W(j)^2 + cx*f == b*W(j);
    eq2 = -r*d*W(j)^2 + r*f + cx*W(j)^3 - cx*e*W(j) == -a*W(j)^2 + c;
    eqs = [eqs eq1 eq2];
end

[a b c d e f] = solve(eqs,variables);

n_coef = [a b c];
d_coef = [1 d e f];
nm = a*s^2 + b*s + c;
dn = s^3 + d*s^2 + e*s + f;

p = nm/dn;

coef = double([a b c d e f]);

% plot resultats
xx = linspace(0,100);
yy = subs(n_0,xx);
plot(xx,yy,'blue')
hold on
yy = subs(d_0,xx);
plot(xx,yy,'blue')
yy = subs(nm,xx);
plot(xx,yy,'green')
yy = subs(dn,xx);
plot(xx,yy,'green')


% PROCES ITERATIU

a = coef(1);
```

```
91  b = coef(2);
92  c = coef(3);
93  d = coef(4);
94  e = coef(5);
95  f = coef(6);
96
97
98  for j=1:num
99
100 %NUMERADOR!!
101
102 nm = a*s^2 + b*s + c;
103 error = nm - n_0;
104
105 % Vectors error
106 E_real = real(double(subs(error,i*W(1:2))));
107 E_complex = imag(double(subs(error,i*W(1))));
108
109 da = [ -1/(x1^2 - x2^2), 1/(x1^2 - x2^2)];
110 db = 1/x1;
111 dc = [ -x2^2/(x1^2 - x2^2), x1^2/(x1^2 - x2^2)];
112
113 a = a - lr * da * transpose(E_real);
114 b = b - lr * db * E_complex;
115 c = c - lr * dc * transpose(E_real);
116
117
118 %DENOMINADOR!
119 dn = s^3 + d*s^2 + e*s + f;
120 error = dn - d_0;
121
122 % Vectors error
123 E_real = real(double(subs(error,i*W(2:3))));
124 E_complex = imag(double(subs(error,i*W(3))));
125
126 dd = [ -1/(x2^2 - x3^2), 1/(x2^2 - x3^2)];
127 de = 1/x3;
128 df = [ -x3^2/(x2^2 - x3^2), x2^2/(x2^2 - x3^2)];
129
130 d = d - lr * dd * transpose(E_real);
131 e = e - lr * de * E_complex;
132 f = f - lr * df * transpose(E_real);
133
134 coef = double([a b c d e f]);
135
136 nm = a*s^2 + b*s + c;
137 dn = s^3 + d*s^2 + e*s + f;
```

```
138 p = nm/dn;
139
140 % Plot resultats
141 yy = subs(nm,xx);
142 plot(xx,yy,'red')
143 yy = subs(dn,xx);
144 plot(xx,yy,'red')
145
146 end
147
148 coef
```

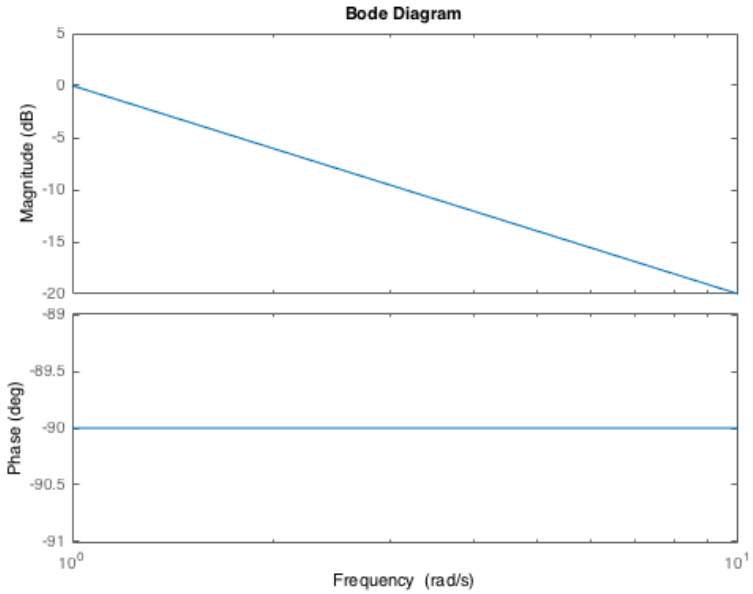


Figure A.1: Frequency response for k = 1

A.3 Fixed LTI ODE operator and variable frequency

If we cannot have a stable LTI ODE operator with variable coefficients, a solution could be to build a transfer function well-designed and stable and then modify the frequency to obtain the desired gain.

We could choose an integrator as our fixed function due to its simplicity,

$$G(s) = \frac{k}{s} \tag{A.1}$$

where k is a real value. The frequency response of this dynamic function is shown in Figure A.1.

By choosing different values for k we can move the gain line to where we want.

We can then choose three differentiated frequency ranges (with completelly different gains) and associate each range to a different problem. By modifying the frequency for each problem we can choose the output gain of the LTI ODE operator.

But we should be able to obtain any real value: positive values and negative values. How can we do so if the 3 gain ranges can't overlap? A way to do it is to limit the possible weight values in the interval $[-1, 1]$. Afterwards, we need to find a correlation between the output gain of the LTI ODE operator and the interval $[-1, 1]$. The easiest way to do this is to make the sinus of the output gain. If we denote the gain returned by the LTI ODE operator as g_i and define $y_i \in [-1, 1]$, then:

$$y_i = sin(g_i) \tag{A.2}$$

because the sinus function always returns values in the interval $[-1, 1]$. In the equation A.2, y_i represents the desired weight value.

So, playing with the constant k, we can find three (or more) different ranges in the frequency response of the LTI ODE operator that satisfies expression (A.2).

Let's put an example to make it clear. Let's imagine a neural network which attempts to solve 3 problems simultaneously. First of all let's associate each problem with the following output gain ranges of the frequency response of the LTI ODE operator:

- Problem 1 : $[12\pi, 10\pi]$ or $[31.52, 29.94]$ dB
- Problem 2 : $[8\pi, 6\pi]$ or $[28, 25.51]$ dB
- Problem 3 : $[4\pi, 2\pi]$ or $[21.98, 15.96]$ dB

If we define our weight function as $G(s) = \frac{12\pi}{s}$ then the following frequency ranges have as output gain values the ranges previously defined. (See Figure A.2, where problem 1 is delimited by red lines, problem two by green lines and problem 3 by black lines)

- Problem 1 : $[1, 1.2]$ Hz
- Problem 2 : $[1.5, 2]$ Hz
- Problem 3 : $[3, 6]$ Hz

In summary, if we associate each problem to the frequency ranges calculated above, we can ensure that the output gain of the LTI-ODE operator will be in the ranges especified. And these gain ranges have been chosen because if we apply the sinus of the gain they return values in the interval $[-1, 1]$.

So, it is possible to build a simple dynamic function that separates three problems in the frequency domain. The problem with this approach is that the frequency associated with each problem is different for every connection between neurons. To physically implement this solution is too complicated.

We have been through all the obvious possibilities to solve the stability problem and no one works. The next step is to simplify again the LTI ODEVNN that we are attempting to solve. The stability conditions are too restrictive and as we have seen, they grow more complicated as we have more coefficients. So, to make the stability conditions simpler the next step is to reduce our LTI ODEVNN to 2 problems instead of 3.

A.4 AND + XOR neural network

A.4.1 main.m

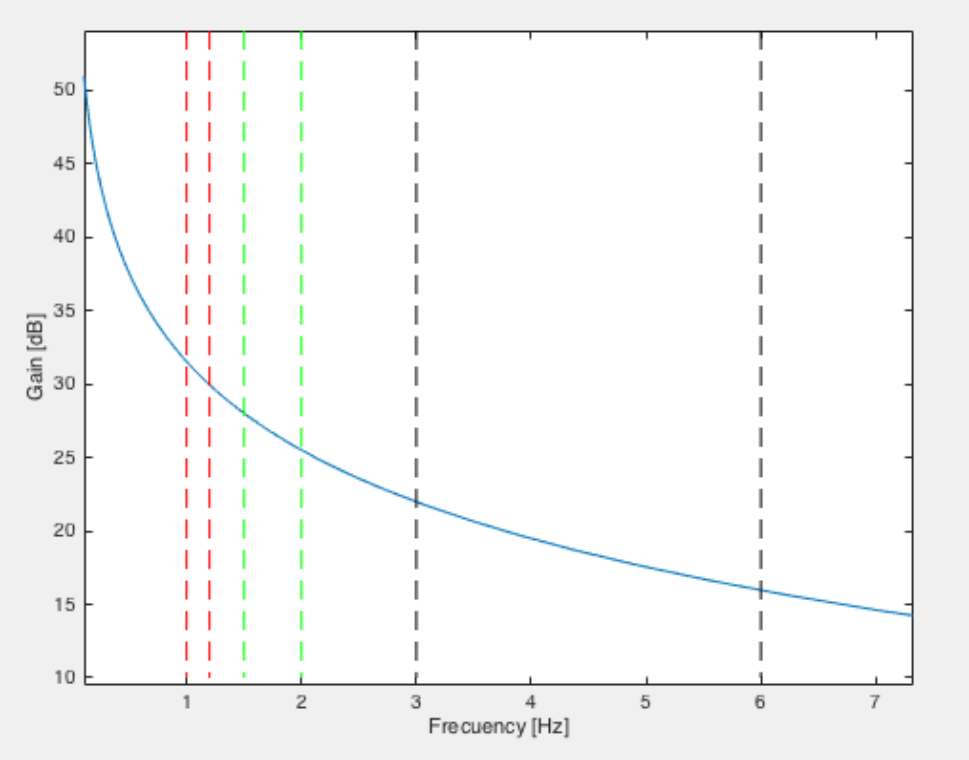


Figure A.2: Gain response of $G(j\omega) = \frac{12\pi}{j\omega}$ with each range of frequency delimited with coloured lines

```
% Frequencies (de l'estil w1 < w2)
f1 = 100; % AND
f2 = 1000; % XOR

w = [2*pi*f1 2*pi*f2];

% Comprovacio de l'estabilitat
estabilitat

% Training data
training_data


% Valors inicials de les funcions pes : 1/(as + b) i els bias entre -1 i
    1
% Estructura d: fila 1: bias; fila 2; input 1; fila 3: input 2
% g: guanys en els rangs associats a cada problema
% p: multiple de 10 associat a cada weight(i,j)
% weights: valors finals del pes

% Neurona 1
g1 = [((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)
    ;...
```

```
22      ((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)];
23  d1 = ab(w, g1, -1 +2.*rand(1,2));
24  p1 = zeros(2,2);
25
26  % Neurona 2
27  g2 = [((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)
        ;...
28      ((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)];
29  d2 = ab(w, g2, -1 +2.*rand(1,2));
30  p2 = zeros(2,2);
31
32  % Neurona 3
33  g3 = [((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)
        ;...
34      ((3*pi)-(2*pi)).*rand + (2*pi), ((3*pi/2)-(pi/2)).*rand + (pi/2)];
35  d3 = ab(w, g3, -1 +2.*rand(1,2));
36  p3 = zeros(2,2);
37
38  % LEARNING PROCESS
39  num = 1000;
40  for n = 1:num
41      for k = 1:data
42          % Forward propagation
43          forwardpropagation
44
45          % Back propagation
46          BackPropagation
47      end
48  end
```

A.4.2 weight.m

```
1  % Associem cada fila a un input i cada columna a un problema
2  function weights = weight(w, d, p)
3      weights = zeros(2,2);
4      for k = 1:2
5          d_tf = tf([1],d(k+1,:));
6          weights(k,1) = cos(bode(d_tf,w(1))) * 10^p(k,1);
7          weights(k,2) = sin(bode(d_tf,w(2))) * 10^p(k,2);
8      end
9  end
```

A.4.3 activation.m

```
function h = activation(z)
    h = [0 0];
    for j = 1:2
        h(j) = 1/(1+exp(-z(j)));
    end
end
```

A.4.4 ab.m

```
% El problema associat a w1 te el rang: 3pi a 2pi (cosinus)
% El problema associat a w2 te el rang: 3pi/2 a pi/2 (sinus)

function d = ab(w,g,bias)
d = ones(3,2);
x1 = w(1)^2;
x2 = w(2)^2;
d(1,:) = bias;
for k = 1:2
    y1 = g(k,1)^(-2);
    y2 = g(k,2)^(-2);
    d(k+1,1) = sqrt((y2-y1)/(x2-x1));
    d(k+1,2) = sqrt((y1*x2-y2*x1)/(x2-x1));
end
end
```

A.4.5 forwardpropagation.m

```
%Forward propagation
% Primera capa (hidden layer)
% Neurona 1
weights_1 = weight(w,d1,p1);
z1 = [d1(1,1) + weights_1(1,1)*x1(k,1) + weights_1(2,1)*x2(k,1), ...
        d1(1,2) + weights_1(1,2)*x1(k,2) + weights_1(2,2)*x2(k,2)];
h1 = activation(z1);

% Neurona 2
weights_2 = weight(w,d2,p2);
z2 = [d2(1,1) + weights_2(1,1)*x1(k,1) + weights_2(2,1)*x2(k,1), ...
        d2(1,2) + weights_2(1,2)*x1(k,2) + weights_2(2,2)*x2(k,2)];
h2 = activation(z2);

```

```
% Segona capa
% Neurona 3
weights_3 = weight(w,d3,p3);
z3 = [d3(1,1) + weights_3(1,1)*h1(1,1) + weights_3(2,1)*h2(1,1), ...
        d3(1,2) + weights_3(1,2)*h1(1,2) + weights_3(2,2)*h2(1,2)];
y = activation(z3);

% inputs de les neurones
x = [x1(k,1), x1(k,2);x2(k,1), x2(k,2)]; % Input primera capa
h = [h1;h2];    % Input segona capa
```

A.4.6 backpropagation.m

```
% Learning rate
lr = 0.7;

% Delta neuron 3
delta_3 = [(y(1)-out(k,1))*y(1)*(1-y(1)), (y(2)-out(k,2))*y(2)*(1-y(2))];

% Delta neuron 1
delta_1 = [delta_3(1) * weights_3(1,1) * h1(1)*(1-h1(1)),delta_3(2) *
    weights_3(1,2) * h1(2)*(1-h1(2))];

% Delta neuron 2
delta_2 = [delta_3(1) * weights_3(2,1) * h2(1)*(1-h2(1)),delta_3(2) *
    weights_3(2,2) * h2(2)*(1-h2(2))];

% New weights

for i = 1:2
    % Bias
    d3(1,i) = d3(1,i) - lr * delta_3(i);
    d2(1,i) = d2(1,i) - lr * delta_2(i);
    d1(1,i) = d1(1,i) - lr * delta_1(i);

    for j = 1:2
        % Inputs
        % Final Weights
        weights_3(j,i) = weights_3(j,i) - lr * h(j,i) * delta_3(i);
        weights_2(j,i) = weights_2(j,i) - lr * x(j,i) * delta_2(i);
        weights_1(j,i) = weights_1(j,i) - lr * x(j,i) * delta_1(i);

        % Power of 10
        if abs(weights_3(j,i)) > 1
```

```
30          p3(j,i) = ceil(log10(abs(weights_3(j,i))));
31          end
32          if abs(weights_2(j,i)) > 1
33          p2(j,i) = ceil(log10(abs(weights_2(j,i))));
34          end
35          if abs(weights_1(j,i)) > 1
36          p1(j,i) = ceil(log10(abs(weights_1(j,i))));
37          end
38      end
39  end
40
41  for j = 1:2
42      % Adapted gains
43      g3(j,1) = acos(weights_3(j,1)/(10^p3(j,1))) + 2*pi;
44      g3(j,2) = pi - asin(weights_3(j,2)/(10^p3(j,2)));
45
46      g2(j,1) = acos(weights_2(j,1)/(10^p2(j,1))) + 2*pi;
47      g2(j,2) = pi - asin(weights_2(j,2)/(10^p2(j,2)));
48
49      g1(j,1) = acos(weights_1(j,1)/(10^p1(j,1))) + 2*pi;
50      g1(j,2) = pi - asin(weights_1(j,2)/(10^p1(j,2)));
51  end
52
53  d3 = ab(w,g3,d3(1,:));
54  d2 = ab(w,g2,d2(1,:));
55  d1 = ab(w,g1,d1(1,:));
```

Appendix B

Static LTI ODE operators

B.1 Minimization with fminsearch

```
function r=minimize(p)

training_data
w1 = [p(1:3);p(4:6)];
w2 = [p(7:9);p(10:12)];
w3 = [p(13:15);p(16:18)];

bias = [-1 -1 -1];

% LEARNING PROCES
dif=0;
for k = 1:4
    forwardpropagation
    y(y>0.8)=1;
    y(y<0.2)=0;
    dif = dif + sum(abs(y-out(k,:)));
end
dif
if dif>0
    r=inf
else
    r=sum((w1(1,2:3)-w1(2,2:3)).^2+(w2(1,2:3)-w2(2,2:3)).^2+(w3(1,2:3)-w3
        (2,2:3)).^2)
end
```

B.2 LTI ODEVNN simultaneously solving 3 problems: MATLAB code

B.2.1 main.m

```
% Training data
training_data

% Number of problems
n = 3;

% Valors inicials
%Structure: row1: neuron 1
%           row2: neuron 2
%           row3: neuron 3
%           column 1: bias problem 1
%           column 2: bias problem 2
%           column 3: bias problem 3
%           column 4: weight input 1
%           column 5: weight input 2

rand('state',sum(100*clock));
w = -1+2.*rand(3,n+2);
w0 = w;

% LEARNING PROCES
num = 1000;
v_bias = 0;
v_weights = 0;
for iter = 1:num
    for k = 1:data
        % Forward propagation
        forwardpropagation

        % Back propagation
        BackPropagation
    end
end
w;
results
```

B.2.2 forwardpropagation.m

```
%Forward propagation
z = zeros(3,n); % Input signal to each neuron
x = zeros(2,n); % Input signal to hidden layer
h = zeros(2,n); % Input signal to last layer
y = zeros(1,n); % Output signal from last neuron

x(1,:) = x1(k,:);
x(2,:) = x2(k,:);


% First layer (hidden layer) (2 neurons)
for j = 1:2
    for i = 1:n
        z(j,i) = w(j,i) + w(j,4)*x(1,i) + w(j,5)*x(2,i);
        h(j,i) = activation(z(j,i));
    end
end

% Second layer (1 neuron)
for i = 1:n
    z(3,i) = w(3,i) + w(3,4)*h(1,i) + w(3,5)*h(2,i);
    y(1,i) = activation(z(3,i));
end
```

B.2.3 backpropagation.m

```
% Learning rate
lr = 0.7;

% Momentum
alfa = 0.1;

% Complexity of each problem
c = [0.1 0.1 2];

delta = zeros(3,n);
% Delta last neuron (second layer)
for i = 1:n
    delta(3,i)= (y(i)-out(k,i))*y(i)*(1-y(i));
end

% Delta neurons from hidden layer (firts layer)
for i = 1:n
    delta(1,i) = delta(3,i) * w(3,4) * h(1,i)*(1-h(1,i));
```

```
19      delta(2,i) = delta(3,i) * w(3,5) * h(2,i)*(1-h(2,i));
20  end
21
22  % New weights
23  % Neuron 3
24  dif3 = h * transpose(c.*delta(3,:));
25  % Neuron 2
26  dif2 = x * transpose(c.*delta(2,:));
27  % Neuron 1
28  dif1 = x * transpose(c.*delta(1,:));
29
30  dif = zeros(3,2);
31  dif(1,:) = transpose(dif1);
32  dif(2,:) = transpose(dif2);
33  dif(3,:) = transpose(dif3);
34
35  %Momentum
36  v_bias = alfa * v_bias + delta;
37  v_weights = alfa * v_weights + dif;
38
39  for j = 1:3
40      for i = 1:n
41          % Bias actualization
42          w(j,i) = w(j,i) - lr * v_bias(j,i);
43      end
44      % Weights actualization
45      w(j,4) = w(j,4) - lr * v_weights(j,1);
46      w(j,5) = w(j,5) - lr * v_weights(j,2);
47  end
```

B.2.4 activation.m

```
1  function h = activation(z)
2          h = 1/(1+exp(-z));
3  end
```

B.3 LTI-ODEVNN simultaneously solving 3 problems: Python code

B.3.1 main.py

```
import numpy as np
import random as r
from functions import*

#Training data
#Input dataset
x1 = np.array([[0,0,0],[0,1,0],[1,0,1],[1,1,1]])
x2 = np.array([[0,0,0],[1,0,1],[0,1,0],[1,1,1]])

#Number of problems
n = 3

#Output dataset
out = np.array([[0,0,0],[1,0,1],[1,0,1],[1,1,0]])
data = out.shape[0]

#Random inicialization for the weights and biases
w = np.zeros((3,5));
for i in range(2):
    for j in range(4):
        w[i,j] = r.uniform(-1,1)


#LEARNING PROCESS
num = 10000;
v_bias = 0;
v_weights = 0;

for iter in range(num):
   for k in range(data):
        #Forward propagation
        (x, h, y) = forwardpropagation(w,x1,x2,n,k)

        #Back propagation
        (w, v_bias, v_weights) = BackPropagation(w,x,h,y,out,v_bias,
           v_weights,n,k)

#Print results
print "Results"
```

```
39 for k in range(data):
40     (x, h, y) = forwardpropagation(w,x1,x2,n,k)
41     print y
```

B.3.2 functions.py

```
1  import numpy as np
2
3  def activation(z):
4      h = 1/(1.0 + np.exp(-z))
5
6      return h
7
8
9  def forwardpropagation(w,x1,x2,n,k):
10
11     #Forward propagation
12     z = np.zeros((3,n)); # Input signal to each neuron
13     x = np.zeros((2,n)); # Input signal to hidden layer
14     h = np.zeros((2,n)); # Input signal to last layer
15     y = np.zeros((1,n)); # Output signal from last neuron
16
17     x[0,:] = x1[k,:]
18     x[1,:] = x2[k,:]
19
20
21     #First layer (hidden layer) (2 neurons)
22     for j in range(2):
23         for i in range(n):
24             z[j,i] = w[j,i] + w[j,3]*x[0,i] + w[j,4]*x[1,i]
25             h[j,i] = activation(z[j,i])
26
27     #Second layer (1 neuron)
28     for i in range(n):
29         z[2,i] = w[2,i] + w[2,3]*h[0,i] + w[2,4]*h[1,i]
30         y[0,i] = activation(z[2,i])
31
32     return (x, h, y)
33
34
35
36 def BackPropagation(w,x,h,y,out,v_bias,v_weights,n,k):
37     # Learning rate
38     lr = 0.7
```

```
39
40      # Momentum
41      alfa = 0.1
42
43      # Complexity of each problem
44      c = np.array([0.1,0.1,2])
45
46      delta = np.zeros((3,n));
47      # Delta last neuron (second layer)
48      for i in range(n):
49          delta[2,i] = (y[0,i]-out[k,i])*y[0,i]*(1-y[0,i])
50
51      # Delta neurons from hidden layer (firts layer)
52      for i in range(n):
53          delta[0,i] = delta[2,i] * w[2,3] * h[0,i]*(1-h[0,i])
54          delta[1,i] = delta[2,i] * w[2,4] * h[1,i]*(1-h[1,i])
55
56      # New weights
57      # Neuron 3
58      dif3 = np.dot(h,np.transpose(c*delta[2,:]))
59      # Neuron 2
60      dif2 = np.dot(x,np.transpose(c*delta[1,:]))
61      # Neuron 1
62      dif1 = np.dot(x,np.transpose(c*delta[0,:]))
63
64      dif = np.zeros((3,2))
65      dif[0,:] = np.transpose(dif1)
66      dif[1,:] = np.transpose(dif2)
67      dif[2,:] = np.transpose(dif3)
68
69      #Momentum
70      v_bias = alfa * v_bias + delta;
71      v_weights = alfa * v_weights + dif;
72
73      for j in range(3):
74          for i in range(n):
75              # Bias actualization
76              w[j,i] = w[j,i] - lr * v_bias[j,i]
77
78          # Weights actualization
79          w[j,3] = w[j,3] - lr * v_weights[j,0]
80          w[j,4] = w[j,4] - lr * v_weights[j,1]
81
82
83      return (w,v_bias,v_weights)
```